Système de gestion de flux pour l'Internet des objets intelligents

Benjamin Billet

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

informatiques mathématiques

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

travaux conduits du 01/02/2012 au 31/01/2015

Spécialité

Informatique - Systèmes distribués

École doctorale Sciences et Technologies de Versailles (STV)

Présentée par

Benjamin BILLET

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

Sujet de la thèse :

SYSTÈME DE GESTION DE FLUX POUR L'INTERNET DES OBJETS INTELLIGENTS

Data Stream Management System for the Future Internet of Things

soutenue le **19 Mars 2015**, devant le jury composé de :

Julien BOURGEOIS (Institut FEMTO-ST, UMR CNRS 6174, FR)	Rapporteur
Nathalie MITTON (Inria Lille-Nord Europe, FR)	Rapporteur
Silvia GIORDANO (University of Applied Science – SUPSI, CH)	Examinateur
Philippe PUCHERAL (Université de Versailles Saint-Quentin-En-Yvelines, FR)	Examinateur
Françoise SAILHAN (Conservatoire National des Arts et Métiers, FR)	Examinateur
Valérie ISSARNY (Inria Paris-Rocquencourt, FR)	Directrice de thèse

À ma famille et à mes amis.

Résumé

L'*Internet des objets* (ou *IdO*) se traduit à l'heure actuelle par l'accroissement du nombre d'objets connectés, c'est-à-dire d'appareils possédant une identité propre et des capacités de calcul et de communication de plus en plus sophistiquées : téléphones, montres, appareils ménagers, etc. Ces objets embarquent un nombre grandissant de capteurs et d'actionneurs leur permettant de mesurer l'environnement et d'agir sur celui-ci, faisant ainsi le lien entre le monde physique et le monde virtuel. Spécifiquement, l'Internet des objets pose plusieurs problèmes, notamment du fait de sa très grande échelle, de sa nature dynamique et de l'hétérogénéité des données et des systèmes qui le composent (appareils puissants/peu puissants, fixes/mobiles, batteries/alimentations continues, etc.). Ces caractéristiques nécessitent des outils et des méthodes idoines pour la réalisation d'applications capables (i) d'extraire des informations utiles depuis les nombreuses sources de données disponibles et (ii) d'interagir aussi bien avec l'environnement, au moyen des actionneurs, qu'avec les utilisateurs, au moyen d'interfaces dédiées.

Dans cette optique, nous défendons la thèse suivante : en raison de la nature continue des données (mesures physiques, évènements, etc.) et leur volume, il est important de considérer (i) les flux comme modèle de données de référence de l'Internet des objets et (ii) le traitement continu comme modèle de calcul privilégié pour transformer ces flux. En outre, étant donné les préoccupations croissantes relatives à la consommation énergétique et au respect de la vie privée, il est préférable de laisser les objets agir au plus près des utilisateurs, si possible de manière autonome, au lieu de déléguer systématiquement l'ensemble des tâches à de grandes entités extérieures telles que le *cloud*.

À cette fin, notre principale contribution porte sur la réalisation d'un système distribué de gestion de flux de données pour l'Internet des objets. Nous réexaminons notamment deux aspects clés du génie logiciel et des systèmes distribués : les architectures de services et le déploiement. Ainsi, nous apportons des solutions (i) pour l'accès aux flux de données sous la forme de services et (ii) pour le déploiement automatique des traitements continus en fonction des caractéristiques des appareils. Ces travaux sont concrétisés sous la forme d'un intergiciel, *Dioptase*, spécifiquement conçu pour être exécuté directement sur les objets et les transformer en fournisseurs génériques de services de calcul et de stockage. Pour valider nos travaux et montrer la faisabilité de notre approche, nous introduisons un prototype de *Dioptase* dont nous évaluons les performances en pratique. De plus, nous montrons que *Dioptase* est une solution viable, capable de s'interfacer avec les systèmes antérieurs de capteurs et d'actionneurs déjà déployés dans l'environnement.

Mots clés : Internet des objets, Gestion de flux de données, Réseau de capteurs et d'actionneurs, Intergiciel, Architecture orientée service.

Abstract

The *Internet of Things* (*IoT*) is currently characterized by an ever-growing number of networked Things, i.e., devices which have their own identity together with advanced computation and networking capabilities : smartphones, smart watches, smart home appliances, etc. In addition, these Things are being equipped with more and more sensors and actuators that enable them to sense and act on their environment, enabling the physical world to be linked with the virtual world. Specifically, the IoT raises many challenges related to its very large scale and high dynamicity, as well as the great heterogeneity of the data and systems involved (e.g., powerful versus resource-constrained devices, mobile versus fixed devices, continuously-powered versus battery-powered devices, etc.). These challenges require new systems and techniques for developing applications that are able to (i) collect data from the numerous data sources of the IoT and (ii) interact both with the environment using the actuators, and with the users using dedicated GUIs.

To this end, we defend the following thesis : given the huge volume of data continuously being produced by sensors (measurements and events), we must consider (i) data streams as the reference data model for the IoT and (ii) continuous processing as the reference computation model for processing these data streams. Moreover, knowing that privacy preservation and energy consumption are increasingly critical concerns, we claim that all the Things should be autonomous and work together in restricted areas as close as possible to the users rather than systematically shifting the computation logic into powerful servers or into the cloud.

For this purpose, our main contribution can be summarized as designing and developing a distributed data stream management system for the IoT. In this context, we revisit two fundamental aspects of software engineering and distributed systems : service-oriented architecture and task deployment. We address the problems of (i) accessing data streams through services and (ii) deploying continuous processing tasks automatically, according to the characteristics of both tasks and devices. This research work lead to the development of a middleware layer called *Dioptase*, designed to run on the Things and abstract them as generic devices that can be dynamically assigned communication, storage and computation tasks according to their available resources. In order to validate the feasability and the relevance of our work, we implemented a prototype of *Dioptase* and evaluated its performance. In addition, we show that *Dioptase* is a realistic solution which can work in cooperation with legacy sensor and actuator networks currently deployed in the environment.

Keywords : Internet of Things, Data Stream Management, Sensor and Actuator Network, Middleware, Service-Oriented Architecture.

Remerciements

Je voudrais remercier les personnes qui ont rendu possibles ces trois années de recherche. En premier lieu ma directrice de thèse, Valérie Issarny, qui m'a aidée ces trois dernières années à ~~m'échapper de ce traquenard~~ aller au bout de ce travail passionnant. Puis les membres de l'équipe Arles-MiMove et ceux de la startup Ambientic, tout spécialement Roberto Speicys-Cardoso et Pierre-Guillaume Raverdy, grâces auxquels j'ai pu faire le stage de fin d'études qui m'a conduit à devenir doctorant.

Je souhaiterais, en outre, remercier mes parents, ma famille et mes amis, dont je me suis éloigné pour venir profiter du climat parisien si caractéristique. Merci d'avoir été là, merci pour vos encouragements et merci de votre soutien dans les moments de doutes. Je sais bien que mon travail vous apparaît comme terriblement obscur vu de loin, mais je tiens à vous rassurer : vu de près aussi.
Une dédicace spéciale à Blandine Bourouliou qui a dépensé beaucoup de temps et d'énergie à corriger ce mémoire.

Enfin, ces remerciements ne seraient pas complets sans citer la créature excentrique qui m'accompagne, et dont l'Histoire retiendra les mots emplis de sagesse : « Greuh ».

Dans la plupart des documents sérieux, le lecteur s'attendra à rencontrer des épigraphes solennelles, érudites ou spirituelles, habituellement destinés à démontrer que le rédacteur est un individu cultivé. Mais qu'y a-t'il au-delà de ces aphorismes prétentieux et de ces apophtegmes[1] *sentencieux, si ce n'est un sentiment diffus de solitude ?*

— Benjamin Billet

1. J'ai gagné mon pari.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

L'Internet des objets est une concrétisation technique de l'*informatique ubiquitaire* où la technologie est intégrée naturellement aux objets du quotidien. Très prometteur, ce concept ouvre la voie vers une multitude de scénarios basés sur l'interconnexion entre le monde physique et le monde virtuel : domotique, e-santé, ville intelligente, logistique, sécurité, etc. Cependant, comme d'autres concepts prometteurs, celui-ci fait face à un certain nombre de problématiques techniques et non techniques qui nécessitent d'être étudiées pour permettre à l'Internet des objets d'atteindre son plein potentiel.

L'*Internet des objets* est un concept concrétisant la vision de l'*informatique ubiquitaire* telle qu'imaginée en 1991 par Mark Weiser [1], où la technologie s'efface peu à peu dans l'environnement des utilisateurs, intégrée naturellement à l'intérieur des objets du quotidien [1]. La technologie n'est plus alors représentée par un objet unique, l'ordinateur personnel, mais se présente au contraire sous la forme d'appareils spécialisés et simples d'emploi, capables de communiquer au travers de plusieurs types de réseaux sans fil : liseuses numériques, télévisions et montres connectées, ordinateurs de bord, téléphones intelligents, etc.

À l'origine, le terme *Internet des objets* a été utilisé pour la première fois en 1999 par Kevin Ashton [2] pour décrire des objets équipés de puces d'identification par radiofréquence (ou puce *RFID*). Chaque objet, identifié de manière unique et universelle, peut alors être rattaché à un ensemble d'informations le concernant, ces dernières étant lisibles par d'autres machines [2]. Caractéristiques, état courant et position sont alors autant de métadonnées échangées entre les objets, formant un nouveau réseau qui leur est dédié : l'Internet des objets.

Le concept a toutefois évolué avec le temps et s'est généralisé vers une approche consistant à connecter un très grand nombre d'objets du quotidien au réseau Internet, les dotant ainsi d'une identité propre et leur permettant, entre autres, d'offrir des services et de collecter des informations de manière autonome. L'objectif ambitieux derrière cette interconnexion est double, consistant en premier lieu dans la mise en place d'une infrastructure de communication machine à machine (*M2M*) à grande échelle de façon à permettre à ces machines de mieux « percevoir » le monde qui les entoure [3]. En second lieu réside une volonté d'offrir les abstractions nécessaires aux êtres humains pour interagir avec ces machines, et par extension avec le monde physique, aussi simplement qu'avec le monde virtuel que nous connaissons aujourd'hui [4]. En d'autres termes, les utilisateurs de l'Internet des objets devraient être en mesure de manipuler l'environnement physique de la même façon qu'ils manipulent aujourd'hui des abstractions de haut niveau, telles que les fichiers et les dossiers, les pages Web et les hyperliens, ou encore les profils et les relations (p. ex. sur les réseaux sociaux).

Jusqu'à présent, le nouveau paradigme qu'est l'Internet des objets émerge des travaux issus de plusieurs communautés scientifiques : les *réseaux de capteurs et d'actionneurs sans fil*, le *Web*, le *cloud computing*, l'*identification par radiofréquence* (*radio-frequency identification*, ou *RFID*) ou encore la *la communication en champ proche* (*near field communication*, ou *NFC*). Cette grande diversité, couplée à la popularité croissante du concept auprès des mondes académique et industriel, fait que, s'il existe effectivement des objectifs communs, l'Internet des objets n'est pas clairement défini, comme le montre la Table 1.1 qui recense

1. Mark Weiser : « The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it. »

2. Kevin Ashton : « RFID and sensor technology enable computers to observe, identify and understand the world. »

The term "Internet of Things" is used as an umbrella keyword for covering various aspects related to the extension of the Internet and the Web into the physical realm, by means of the widespread deployment of spatially distributed devices with embedded identification, sensing and/or actuation capabilities. Internet of Things envisions a future in which digital and physical entities can be linked, by means of appropriate information and communication technologies, to enable a whole new class of applications and services [4].
An "Internet of Things" means "a world-wide network of interconnected objects uniquely addressable, based on standard communication protocols", or, more widely : "Things having identities and virtual personalities operating in smart spaces using intelligent interfaces to connect and communicate within social, environmental, and user contexts" [5].
A global network infrastructure, linking physical and virtual objects through the exploitation of data capture and communication capabilities. This infrastructure includes existing and evolving Internet and network developments. It will offer specific object-identification, sensor and connection capability as the basis for the development of independent cooperative services and applications. These will be characterized by a high degree of autonomous data capture, event transfer, network connectivity and interoperability [6].
The basic idea of this concept is the pervasive presence around us of a variety of things or objects – such as Radio-Frequency IDentification (RFID) tags, sensors, actuators, mobile phones, etc. – which, through unique addressing schemes, are able to interact with each other and cooperate with their neighbors to reach common goals [7].

Table 1.1 – Quelques définitions de l'Internet des objets.

quelques définitions récentes extraites de la littérature. En combinant ces définitions, on retrouve toutefois les aspects que nous avons évoqués plus haut : identité, autonomie, interconnexion et interaction avec l'environnement (capteurs et actionneurs).

À noter que le terme « systèmes cyberphysiques » (*cyber physical system*, ou *CPS*) est parfois utilisé pour désigner des concepts proches de l'Internet des objets. Si l'on en croit les définitions données dans la littérature, les systèmes cyberphysiques partagent le même objectif de mise en relation du monde physique et du monde virtuel (ou *cyberworld*) [8]. L'existence des deux termes pourrait toutefois s'expliquer par la définition première de l'Internet des objets, qui portait exclusivement sur les objets identifiés par une puce *RFID*. Aussi, par la suite, nous nous restreindrons uniquement à l'utilisation du terme Internet des objets.

Qu'est-ce qu'un objet ?

Un *objet* est, avant toute chose, une entité physique ; par exemple, un livre, une voiture, une machine à café électrique ou un téléphone mobile. Dans le contexte précis de l'Internet des objets, et ce quelle que soit la vision, cet objet possède au minimum un *identifiant unique* attaché à une *identité* [7] exprimant d'une part ses *propriétés* immuables (type, couleur, poids, etc.) et son *état* c'est-à-dire l'ensemble de ses caractéristiques pouvant évoluer au cours du temps (position, niveau de batterie, etc.).

De nombreuses définitions s'accordent à dire qu'un objet possède des capacités de calcul, d'acquisition (capteur) et d'action (actionneur) [4, 6, 9], cependant cette définition

exclut les objets inertes identifiés par *RFID*. En effet, une puce *RFID* classique ne peut pas être considérée comme un dispositif de calcul, celle-ci se résumant à un identifiant stocké dans une mémoire. De même, si l'on considère les cas extrêmes, un simple code-barre peut être employé pour identifier un objet, ce dernier étant exempt d'une quelconque partie électronique ; un livre et son code *ISBN*, par exemple. Ainsi, il est raisonnable de considérer que l'Internet des objets est composé d'objets *actifs*, capables d'accomplir des calculs, d'effectuer des mesures sur l'environnement ou d'influer sur celui-ci, et d'objets *passifs* qui n'ont pas d'autres aptitudes que celles d'être suivis et détectés par des objets actifs. Par extension, l'identité d'un objet passif n'est pas directement stockée dans celui-ci, à l'exception de l'identifiant, et nécessite l'utilisation d'une infrastructure tierce capable de stocker ces informations. Au contraire, un objet actif peut stocker tout ou partie de son identité et échanger directement ces informations avec d'autres objets actifs. Cependant, cette capacité à stocker sa propre identité n'est pas obligatoire pour un objet actif et dépend (i) de ses ressources matérielles, notamment la mémoire, et (ii) de la complexité et du volume de ladite identité.

De manière générale, les ressources matérielles des objets actifs ont une importance cruciale en cela qu'elles définissent quels traitements pourront être effectués par l'objet. Par exemple, un algorithme de reconnaissance faciale nécessite plus de ressources qu'un algorithme de recherche d'éléments dans un ensemble ordonné, ce dernier nécessitant lui-même plus de ressources qu'une simple opération arithmétique telle que l'addition. Par ailleurs, les systèmes embarqués offrent souvent un ensemble d'opérations spécifiques implémentées directement au niveau du matériel, notamment sous la forme de circuits logiques programmables. Ainsi, même des objets aux ressources *CPU* et mémoire limitées peuvent être dotés d'un module matériel de sécurité (*hardware security module*) ou d'une puce d'encodage ou de décodage vidéo. De la même façon, les objets actifs sont aussi des objets communicants et, de fait, embarquent diverses interfaces leur permettant d'échanger des informations sur divers réseaux *filaires* ou *sans fil*.

Au-delà des ressources matérielles, les objets physiques possèdent des caractéristiques variées dues à leurs usages. Certains objets sont *mobiles* et, de fait, caractérisés par une position qui évolue au cours du temps. Il peut s'agir d'objets *transportables*, tels qu'un téléphone mobile, un livre ou des vêtements, ou d'objets *mobiles autonomes* dotés de capacités motrices, tels qu'une voiture, un drone ou un animal domestique [3]. À l'inverse, de nombreux objets sont fixes la plupart du temps : réfrigérateurs, meubles, compteurs électriques, etc. De la même façon, certains objets sont *alimentés en continu* depuis une source permanente d'énergie électrique (typiquement, les appareils électroménagers) tandis que d'autres sont *alimentés par une batterie* et possèdent donc une durée de vie limitée en fonction de celle-ci. Ce facteur a une influence variable selon si l'objet est *facilement rechargeable*, comme c'est le cas avec un téléphone mobile ou un baladeur

3. Un animal est un objet au sens restreint d'ensemble de caractéristiques mesurables et évoluant au cours du temps.

numérique, ou au contraire abandonné sur le terrain, par exemple dans le cadre d'études des mouvements migratoires d'une population animale. De la même façon, certains objets sont en mesure d'extraire de l'énergie depuis leur environnement (*energy harvesting*) par exemple en tirant parti de l'énergie solaire ou d'un champ électromagnétique. En outre, certains appareils dotés de capacités motrices sont capables de rechercher des points de rechargement et de s'y rendre sans intervention humaine. C'est notamment le cas des aspirateurs robots.

Qu'est-ce que l'Internet des objets ?

Le concept d'objet étant posé, les différentes définitions présentées Table 1.1 et la littérature que nous avons étudiée à ce sujet nous conduisent à conclure que l'Internet des objets est une infrastructure globale permettant notamment :

- aux objets actifs d'échanger des informations ou d'en collecter à propos des objets passifs, ce qui rejoint les définitions d'un réseau *M2M* mondial [4] ;
- de stocker et de rendre accessible l'identité des objets, les informations produites par ces derniers et toutes les connaissances nécessaires pour que les objets gagnent en autonomie, ce au travers de représentations, de structures et de formats de données manipulables par les machines [5] ;
- aux êtres humains d'accéder à ces informations et d'interagir naturellement avec les objets, au niveau local et au niveau global, ce qui rejoint les définitions sur la connexion du monde physique et du monde virtuel [6].

Au-delà de ces trois rôles, l'Internet des objets est influencé par les différents scénarios d'utilisation qui sont considérés aujourd'hui par les mondes scientifique et industriel [4, 7, 14, 15]. Quelques exemples courants sont présentés dans la Table 1.2 à titre d'illustration. Chacun d'entre eux possède ses propres objectifs et ses propres contraintes, d'où l'intérêt de considérer l'Internet des objets non pas sous le prisme d'une application spécifique, mais comme la combinaison des besoins d'une multitude de scénarios. Ces scénarios peuvent prendre place dans des *espaces privés*, au sein desquels les interactions avec l'extérieur sont strictement contrôlées, ou dans des *espaces publics* ouverts à tous (par exemple, la détection des places de parking vides), avec toute la gamme de nuances qui peut exister entre les deux. En effet, un logement intelligent qui régule automatiquement la luminosité ou la température est un espace strictement privé tandis qu'un parking public qui indique aux automobilistes quelles sont les places libres les plus proches est un espace strictement public. Des interactions diverses vont pouvoir exister entre les différents espaces au travers d'interfaces spécialisées : par exemple, une personne physique ou morale peut choisir de diffuser certaines informations hors de ses espaces privés.

De la même façon, certains scénarios impliquent la participation directe des utilisateurs ; par exemple, une application mobile où les utilisateurs évaluent subjectivement la densité de population d'une station de métro ou la propreté d'un lieu public. D'autres cas

Suivi et logistique	Rendre possible le suivi dans le temps et l'espace de n'importe quel objet, pour peu que celui-ci soit identifié (p. ex. avec une étiquette *RFID*). Ce scénario est aujourd'hui implémenté par les industriels pour le suivi des objets dans une chaîne de production (fabrication, transport, etc.) ou dans un stock, le suivi d'objets perdus ou volés, ou encore la classification de documents [7].
Soins médicaux et aide à la personne	Utiliser des capteurs pour obtenir des informations en temps réel sur l'état des patients (rythme cardiaque, pression sanguine, etc.), même lorsque ceux-ci sont en extérieur (capteurs portatifs), pour pouvoir réagir rapidement en cas d'anomalie [10].
Environnements intelligents	Autoréguler les paramètres de l'environnement dans les logements et les bureaux (température, luminosité, humidité, consommation énergétique, etc.) et contrôler l'accès à certaines ressources en fonction du contexte (p. ex. présence ou non d'un responsable). Dans le cadre des villes intelligentes, analyser en temps réel les différents paramètres (bruit, pollution, trafic routier) et connecter les différents éléments de l'infrastructure urbaine pour en améliorer la gestion [11].
***Sensing* social, *sensing* partagé et *sensing* participatif**	Utiliser les capteurs portatifs pour collecter un ensemble d'informations concernant, par exemple, la position, l'activité en cours (courir, faire du vélo, etc.), l'humeur ou l'environnement (bruyant, peuplé, pluie, etc.) d'un individu dans le but de les partager automatiquement avec des groupes sociaux [12]. Par extension, permettre aux utilisateurs et aux organisations de partager les informations de leurs systèmes à grande échelle pour contribuer collaborativement à de nombreuses tâches d'analyse [13] : trafic routier, flux de population, conditions climatiques, etc.

Table 1.2 – Quelques scénarios pour l'Internet des objets.

d'utilisations se basent sur le partage de certains objets appartenant aux utilisateurs, dans une optique collaborative. Par exemple, les voitures étant de plus en plus équipées d'un GPS, il est possible de proposer aux utilisateurs de contribuer à l'analyse du trafic routier.

1.1 Problématiques posées par l'Internet des objets

Pour permettre à l'Internet des objets d'atteindre son plein potentiel et concrétiser les scénarios qui en découlent, plusieurs aspects doivent être étudiés. En effet, les concepts prometteurs imaginés par la communauté scientifique nécessitent de résoudre un certain nombre de problématiques : grande échelle, hétérogénéité, impact du monde physique (grands flux continus de données, variabilité de l'environnement), sécurité des biens et des personnes, et respect de la vie privée des utilisateurs.

1.1.1 Échelle de l'Internet des objets

De toutes les difficultés posées par l'Internet des objets, son échelle mondiale et le très grand nombre d'objets impliqués sont les plus importantes, et ce même si l'on exclut

Adressage et nommage	Le très grand nombre d'objets nécessite (i) un espace d'adressage important et (ii) augmente significativement la quantité d'informations que les serveurs de noms doivent stocker pour assurer leur rôle d'association entre les noms d'objet et leurs adresses.
Découverte	Enregistrer et rechercher des objets par leur nom ou par leurs caractéristiques est fondamental pour la réalisation des scénarios où les objets ne sont pas connus à l'avance. Cependant, stocker et parcourir l'ensemble des objets de manière centralisée n'est pas envisageable à une telle échelle.
Accès	La multitude d'objets différents complexifie le rapatriement des données (*data collection*), tandis que les grands volumes de données complexifient leur traitement (*data mining* et *data aggregation*).

Table 1.3 – Résumé de l'impact de l'échelle sur l'Internet des objets.

les objets passifs qui n'y sont connectés que par procuration. Il paraît raisonnable, en ce qui concerne le réseau de communication global proprement dit, de considérer le réseau Internet comme un candidat prometteur [4]. En effet, celui-ci est déjà massivement déployé à l'échelle mondiale et interconnecte plusieurs milliards de machines diverses et variées. En cela, Internet est le plus grand réseau connu jamais déployé, et paraît donc une bonne base pour l'Internet des objets. Par ailleurs, on estime qu'aujourd'hui une portion non négligeable des connexions sur Internet se font déjà entre objets connectés, et que ceux-ci sont au nombre de 15 milliards environ [16]. D'après la progression actuelle, ce nombre pourrait augmenter et atteindre jusqu'à 100 milliards d'objets d'ici 2020 [17], avec un trafic Internet global qui se compterait en zettaoctets [4] (10^{21} octets).

Les différents problèmes liés à l'échelle de l'Internet des objets sont résumés dans la Table 1.3. On y trouve en premier lieu les problèmes relatifs à l'adressage et l'identification des objets, que l'on espère cependant résoudre au moyen de technologies existantes :

— Le protocole *IPv6* [18] pour l'adressage des objets.

— Les codes de produits électroniques (*electronic code product*, ou *EPC*) [19], les *IRI*[5] (*internationalized resource identifier*) [21] ou les identifiants universels uniques (*universally unique identifier*, ou *UUID*) [22] pour l'identification unique des objets.

Toutefois, au-delà du simple adressage, d'autres problématiques se posent quant à la gestion de l'identité des objets et la découverte de ces derniers. De manière générale, le processus de découverte permet aux machines de prendre connaissance de l'existence d'autres machines, soit (i) en émettant un message à destination de toutes les machines accessibles directement sans routage (découverte locale), soit (ii) en contactant un annuaire externe, c'est-à-dire une base de données stockant les identités d'objets [23]. Découvrir les objets est fondamental dans le cadre de certains scénarios de l'Internet des objets,

4. *http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html* (accédé le 08/07/2014).

5. Un *URI* est typiquement limité aux caractères *ASCII*. Il en existe cependant une version internationalisée, appelée *IRI* (*internationalized uniform identifier*), supportant l'ensemble des caractères *Unicode* [20].

typiquement lorsque ce dernier utilise des capteurs qui ne sont pas explicitement connus au préalable. Aussi, grâce à l'infrastructure de découverte [7], il doit être possible de :

— récupérer et mettre à jour l'identité d'un objet à partir de son adresse ou de son identifiant ;

— récupérer les adresses des objets dont l'identité satisfait certaines contraintes.

La première opération s'apparente à ce que nous connaissons aujourd'hui avec le *système de noms de domaine* (*domain name system*, ou *DNS*) [24] du réseau Internet : un annuaire hiérarchique permettant de retrouver l'adresse *IP* d'une machine à partir de son nom d'hôte (p. ex. « www.inria.fr »). De la même façon, le concept de *système de noms d'objet* (*object name system*, ou *ONS*) [7] peut être introduit pour retrouver l'identité d'un objet à partir de son adresse ou de son identifiant. Si un *ONS* conçu sur le modèle du *DNS* paraît envisageable à l'échelle de l'Internet des objets, l'opération inverse consistant à retrouver les adresses des objets à partir de leurs caractéristiques est cependant plus complexe. En effet, certains paramètres de l'identité peuvent changer au cours du temps, le plus courant étant la position, ce qui complexifie la synchronisation dans le cas d'un *ONS* inverse distribué. Toutefois, à une telle échelle, une approche centralisée est difficilement envisageable étant donné la quantité d'identités à stocker [23].

En plus d'avoir un impact sur la découverte, l'échelle de l'Internet des objets complexifie l'accès et la collecte des informations produites par les objets. Par exemple, si l'on considère un scénario consistant à analyser la pollution d'une ville, l'ensemble des capteurs disponibles pourrait être très grand, car comprenant les capteurs fixes installés par la ville et les éventuels capteurs mis à disposition par les citoyens. Accéder séparément à chaque capteur pour y récupérer un flux de mesures conduirait à gaspiller inutilement les ressources énergétiques des capteurs et à saturer les canaux de communication. Par ailleurs, le problème est démultiplié lorsque la taille des données augmente, comme c'est le cas pour les systèmes publics de vidéosurveillance. Ces grands volumes de données, issus d'un grand nombre de sources différentes, posent un problème en ce qui concerne leur analyse (filtrage, agrégation, recherche, etc.) et leur stockage [16], et requièrent des approches et des techniques spécifiques de traitement continu en temps réel.

1.1.2 Hétérogénéité de l'Internet des objets

Lorsque nous nous sommes précédemment posé la question « qu'est-ce qu'un objet ? », nous avons pu remarquer que tous les objets n'étaient pas égaux. En effet, conformément aux usages pour lesquels ils ont été conçus, les propriétés et les capacités des objets varient significativement, contribuant à faire de l'Internet des objets un écosystème certes riche, mais aussi hétérogène, les effets de cette hétérogénéité étant résumés dans la Table 1.4. Aux caractéristiques des objets correspondent des contraintes qui conditionnent la façon dont ces objets sont utilisés et interagissent avec leur environnement. Par exemple, les objets mobiles sont plus susceptibles de souffrir d'une connectivité intermittente que

Hétérogénéité fonctionnelle	Les objets possèdent des capacités spécifiques (statique ou mobile, alimenté en continu ou par une batterie, ressources matérielles, capteurs et actionneurs disponibles, etc.) et à chacune d'elle correspond des contraintes particulières (connectivité intermittente, durée de vie, tâches réalisables, etc.). Différentes approches et techniques doivent être considérées pour gérer les objets en fonction de leurs contraintes et de leurs différences propres.
Hétérogénéité technique	Les technologies matérielles et logicielles utilisées pour construire les objets sont multiples et compromettent l'idéal de collaboration autonome entre objets. De plus, le développement d'applications est complexifié, nécessitant des développeurs des connaissances spécifiques sur le fonctionnement de chaque objet.

Table 1.4 – Résumé de l'impact de l'hétérogénéité sur l'Internet des objets.

les objets statiques, les objets alimentés par une batterie sont limités par leur durée de vie et les objets possédant de faibles ressources matérielles sont restreints à des tâches simples. De la même façon, les objets embarquent différents capteurs et actionneurs conformément à leurs fonctions, ce qui a un impact direct sur leurs aptitudes à mesurer et à influer sur leur environnement. Sachant que l'Internet des objets est un paradigme dans lequel les différents objets sont interconnectés et collaborent dynamiquement entre eux, cette *hétérogénéité fonctionnelle* doit être prise en compte. De manière générale, les objets devraient pouvoir interagir en toutes circonstances, quelles que soient les contraintes posées par leurs caractéristiques et leur environnement [25].

En plus d'être hétérogène au niveau fonctionnel, l'Internet des objets est directement affecté par la diversité des composants matériels et logiciels utilisés pour construire les objets. En effet, en fonction des constructeurs, les objets n'utilisent pas les mêmes systèmes d'exploitation, ne possèdent pas les mêmes interfaces de communication et ne respectent pas les mêmes formats de données, ce qui conduit à une *hétérogénéité technique* significative. Si un effort de standardisation est mené par différents organismes comme l'*IEEE* , l'*IETF* ou le *W3C*[6], de nombreuses technologies propriétaires restent couramment utilisées [26,27]. En outre, les différentes façons de programmer les systèmes complexifient le développement d'applications à grande échelle en obligeant les développeurs à posséder des connaissances sur le fonctionnement des objets de chaque constructeur. Concernant l'autonomie des objets, les multiples interfaces de communication et protocoles utilisés, lorsque ceux-ci sont incompatibles, limitent les interactions naturelles entre les appareils et contribuent à former des ilots d'objets isolés les uns des autres [23]. Enfin, les formats utilisés pour nommer les objets, décrire leurs identités et représenter les données que leurs capteurs produisent sont nombreux, d'où d'importantes difficultés pour deux objets d'échanger des informations « compréhensibles ».

Les effets négatifs de ces deux formes d'hétérogénéité, décuplés par la grande échelle et la popularité croissante de l'Internet des objets auprès des constructeurs, peuvent être

6. *http://www.ietf.org*, *http://standards.ieee.org*, *http://www.w3.org* (toutes accédées le 18/06/2014).

Flux de données	Les informations produites par les capteurs sont naturellement liées au temps, sous la forme de flux de mesures ou d'évènements. Cette particularité s'oppose radicalement aux approches classiques basées sur des ensembles de données finis, et nécessite une réflexion différente en ce qui concerne la représentation des données (*data model*) et leur traitement (*computation model*).
Capteurs	Les capteurs sont connus pour produire régulièrement des mesures erronées, imprécises ou incomplètes, sans qu'il soit possible de prédire précisément à quel moment celles-ci vont apparaître (*transient faults*) [27]. Des techniques spécifiques de détection, de correction ou d'atténuation d'erreurs doivent être mises en œuvre, tout particulièrement dans un environnement multicapteur comme l'Internet des objets.
Variabilité	Le monde physique est un environnement changeant, spécifiquement dans le cadre des objets mobiles qui doivent composer avec leurs limites en énergie, la connectivité intermittente et les mouvements de leurs utilisateurs. Aussi, les objets doivent pouvoir s'adapter aux changements qui surviennent au cours du temps, conformément aux besoins des utilisateurs.

Table 1.5 – Résumé de l'impact du monde physique sur l'Internet des objets.

atténués en agissant sur plusieurs fronts. Tout d'abord, pour lutter contre l'hétérogénéité technique, l'effort de standardisation technologique doit être mené à son terme, notamment au travers de la convergence des technologies dédiées aux systèmes limités en ressources vers celles du réseau Internet actuel [26]. Ensuite, des couches d'abstraction logicielles doivent être introduites pour gérer l'hétérogénéité fonctionnelle de l'Internet des objets, couplées à l'utilisation de formats ouverts et interopérables. Enfin, il paraît utile de rendre les objets plus autonomes, notamment en leur permettant de manipuler des connaissances grâce aux technologies de description de la sémantique. Ainsi, en raisonnant automatiquement à partir des concepts, les objets seraient plus à même de s'adapter dynamiquement à leur environnement et aux contraintes fonctionnelles et techniques, lorsque cela est possible [25].

1.1.3 Influence du monde physique sur l'Internet des objets

L'Internet des objets est fondamentalement influencé par les caractéristiques du monde physique et des outils qui permettent de le mesurer, comme résumé dans la Table 1.5. Le monde physique est un environnement qui évolue naturellement au cours du temps, d'une part parce que des objets y sont ajoutés ou retirés continuellement et, d'autre part, du fait de la mobilité des êtres humains et de certains de leurs objets. Globalement, de nombreux objets sont amenés à découvrir dynamiquement leur environnement et à communiquer de manière opportuniste avec d'autres objets, c'est-à-dire initier des échanges à un instant t avec des objets proches qui ne seront peut-être plus disponibles à l'instant $t + 1$. Concrètement, la mobilité et les interactions à la volée induisent une topologie dynamique du réseau, où certains objets bougent, apparaissent et disparaissent au gré de leurs batteries, des mouvements de leurs utilisateurs et des infrastructures de

communication sans fil disponibles (connectivité intermittente). Les objets doivent donc être en mesure de s'adapter dynamiquement aux changements qui surviennent au cours du temps, aussi bien en ce qui concerne leurs propres états que leur environnement. Aussi, conformément aux besoins exprimés par les utilisateurs, les objets devraient posséder les moyens nécessaires pour réagir dynamiquement en fonction des situations, que ce soit en collaborant avec d'autres objets ou en appliquant des techniques appropriées.

Au-delà de l'autonomie des objets, relier le monde physique au monde virtuel implique de tenir compte des particularités et des limites fondamentales des systèmes qui permettent d'interagir avec ce monde physique : les capteurs et les actionneurs. De manière générale, les capteurs produisent soit des mesures régulières d'un phénomène donné (p. ex. capteur de température) soit des évènements (p. ex. capteur de présence) [27]. Ces informations sont ensuite transformées et consommées par d'autres objets, dont les actionneurs qui sont généralement placés en bout de chaîne. On constate que les deux familles d'information générées par les capteurs sont dépendantes du temps :

— les mesures régulières représentent l'évolution d'un phénomène au cours du temps ;

— les évènements correspondent à un fait limité dans le temps.

Aussi, qu'il s'agisse de flux d'échantillons ou de flux d'évènements, un capteur produit des données en continu qui doivent être traitées au fur et à mesure. En effet, il n'est pas envisageable d'attendre la fin du flux avant de commencer à effectuer des opérations, étant donné que (i) l'on ignore quand le flux se terminera (infinité potentielle), (ii) les données produites perdent de leur intérêt au cours du temps et (iii) nous ne disposons pas forcément de l'espace de stockage nécessaire.De manière générale, n'importe quelle information évoluant au cours du temps peut être représentée comme un flux : trafic réseau, valeur d'une action en bourse, journal d'erreur, historique d'appels téléphoniques, etc. En outre, même sur le Web, un nombre croissant de services proposent une approche sous forme de flux de données, comme les *timelines* des réseaux sociaux, les flux de syndication [7], ou encore l'arbre des versions dans les outils de travail collaboratif ; wikis et serveurs de version, par exemple.

Contrairement aux ensembles finis de données classiques que l'on rencontre habituellement dans les bases de données (les tables) ou sur le Web (les pages), les flux de données sont produits, traités ou consommés en continu, élément après élément. Peu d'informations sont connues à priori lorsque l'on traite un flux de données : celui-ci est potentiellement infini et son débit peut être variable, tout particulièrement dans le cas des évènements dont l'occurrence, la durée ou la taille ne sont pas forcément prédictibles. À l'échelle de l'Internet des objets, certains flux peuvent avoir un débit très important, à tel point que les données produites ne peuvent pas être stockées dans leur intégralité. Dans un tel scénario, les nouveaux éléments produits dans le flux doivent être traités en une

7. Sur le Web, la syndication désigne un flux d'information mis à disposition par un site Web à destination d'autres sites Web (liste du contenu récemment ajouté, par exemple). Les technologies de syndication Web les plus connues sont les *flux RSS* et les *flux Atom* [28].

seule passe puis immédiatement supprimés de la mémoire [29]. Pour pouvoir faire face à cette contrainte, des algorithmes spécialement conçus à cet effet doivent être employés. En outre, la nature continue des données et des traitements nécessite des langages et des paradigmes de programmation capables de fournir les abstractions appropriées aux développeurs. Bien qu'il soit possible d'écrire des programmes de traitement continu au moyen des langages généralistes courants, l'absence d'abstractions appropriées induit un décalage entre ce qui doit être exprimé par les développeurs et ce que ces langages peuvent exprimer naturellement [30].

L'Internet des objets, du fait de son utilisation massive de capteurs, doit aussi faire face aux limites techniques de ces derniers, connus pour introduire des erreurs de mesure [31]. Plusieurs types d'erreurs peuvent être rencontrés [31, 32], notamment :

— un bruit aléatoire (*noise*) qui affecte toute une série de mesures ;

— une variation inattendue entre deux points (*outlier*) ou dans un ensemble de points (*spike*) ;

— une succession de mesures identiques, dont la valeur est soit anormalement basse ou haute, soit anormalement constante.

Il est généralement impossible de prédire quand ces erreurs vont survenir (*transient faults*) [27] et leurs causes sont difficiles à déterminer de manière spécifique : il peut s'agir de défaillances matérielles, comme un court-circuit ou un capteur endommagé, d'un effet dû au déchargement de la batterie (introduction de bruit), d'une défaillance logicielle ou d'une erreur de calibration [31]. Ces dernières sont très répandues et peuvent corrompre les résultats de différentes manières, par exemple en introduisant un décalage constant (*offset fault*), un gain constant (*gain fault*) ou une variation de ces deux propriétés au cours du temps (*drift fault*). Elles sont en outre difficiles à détecter, car l'ensemble des mesures est affecté et, de fait, conserve une certaine cohérence vis-à-vis du phénomène mesuré. Par exemple, un décalage constant sur un capteur de température ne perturberait pas la cohérence du cycle jour-nuit [31].

La lecture des puces *RFID* et la communication en champ proche, bien que n'effectuant pas de mesures à proprement parler, souffrent des mêmes perturbations et erreurs de mesure [33]. Les lecteurs échouent régulièrement à détecter les objets à proximité (environ 30% d'échec) ou produisent des détections erratiques, par exemple sous la forme de multiples détections dans un court intervalle de temps par un ou plusieurs lecteurs [34]. De la même façon, les technologies de communication en champ proche (*NFC*) sont sujettes à de fréquentes déconnexions et un fort taux de pertes [35].

La détection de ces erreurs et leur correction est une tâche complexe en temps normal, mais devient critique à l'échelle de l'Internet des objets où le volume de données est tel qu'il n'est ni souhaitable ni forcément possible de les stocker pour les parcourir plusieurs fois. De la même façon que la nature temporelle des informations produites par les capteurs nécessite que l'on s'intéresse à une représentation des données sous forme de

Vulnérabilité	L'impossibilité d'utiliser les technologies modernes de sécurité (chiffrement, authentification, échange de clé, signature, etc.) rend les objets vulnérables aux attaques informatiques ce qui, en raison de leurs capacités à influer sur le monde physique, représente un danger pour les biens et les personnes.
Surveillance de masse	En s'intégrant naturellement à l'environnement, les objets peuvent mesurer n'importe quelle information sans qu'il soit possible de savoir où, quand et par qui sont collectées ces données. En effet, même en l'absence d'intentions malveillantes de la part des propriétaires d'objets, l'utilisation massive des réseaux sans fil facilite les écoutes clandestines. Enfin, l'identification unique des objets permet de dresser des profils très détaillés de leurs propriétaires et de les suivre à grande échelle.

Table 1.6 – Résumé des problèmes de sécurité et de vie privée dans l'Internet des objets.

flux, les processus de détection et de correction des erreurs doivent eux aussi pouvoir être appliqués en continu.

En outre, l'hétérogénéité joue un rôle ici aussi, étant donné que d'une famille de capteurs à une autre les propriétés de mesure peuvent varier : précision, résolution, sensibilité, plage de mesure, temps de réponse, etc. De manière analogue, les probabilités d'occurrence de certaines erreurs vont être dépendantes des technologies de capteurs utilisées.

1.1.4 Sécurité et vie privée

Si l'Internet des objets introduit plusieurs cas d'utilisation prometteurs, il pose aussi plusieurs problèmes quant à la sécurité et à la vie privée, comme résumé dans la Table 1.6. En effet, les objets étant des entités physiques possédant les capacités nécessaires pour influer sur leur environnement, les dégâts provoqués par une attaque informatique dans un tel contexte ont un impact bien plus important que ceux causés par les intrusions, les défigurations de site, les vols de données ou les dénis de service que nous connaissons à l'heure actuelle. À titre d'exemple, on citera *Stuxnet*[8], un ver sophistiqué, découvert en 2010, qui s'attaque spécifiquement aux systèmes de contrôle et d'acquisition de données (*supervisory control and data acquisition*, ou *SCADA*) utilisés, entre autres, pour contrôler des infrastructures industrielles (privées, militaires et civiles) de premier plan : centrales nucléaires, oléoducs et gazoducs, aéroports, etc.

Concrètement, l'Internet des objets souffre de plusieurs vulnérabilités inhérentes aux technologies utilisées. En effet, de nombreux objets possèdent des capacités matérielles réduites, voire quasiment inexistantes, et, de fait, ne peuvent pas directement mettre en œuvre des techniques de sécurisation modernes, ces dernières nécessitant des ressources de calcul conséquentes [7]. De la même façon, le coût énergétique élevé des communications sans fil complexifie les processus d'authentification des objets au travers

8. *http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf* (accédé le 21/07/2014).

d'infrastructures dédiées, du fait de l'échange de multiples messages entre les appareils et les serveurs d'authentification. L'échange de clés entre les différents objets pose des problèmes similaires, aussi bien en matière de calculs (génération des clés) qu'en matière de communication (échange des clés). Pour finir, la préservation de l'intégrité des données, c'est-à-dire la garantie qu'un adversaire ne puisse pas modifier de données sans que cela soit détecté par le système, est difficile à assurer en l'absence de chiffrement fort. Par ailleurs, les objets sont vulnérables physiquement, étant le plus souvent déployés dans un environnement sans surveillance particulière. Un attaquant pourrait, notamment, altérer leur mémoire en accédant directement à leur matériel ou aux interfaces d'entrée-sortie disponibles. En outre, l'utilisation des technologies de communication sans fil facilite d'une part la mise en place d'écoutes clandestines [36] et, d'autre part, les attaques par déni de service en perturbant le signal radio.

Étant donné l'intégration naturelle des objets dans l'environnement, de nombreuses informations peuvent être collectées à l'insu des utilisateurs, aussi bien par le propriétaire des objets à proximité que par d'éventuels attaquants. De manière générale, il n'est plus possible de savoir *où, quand, pourquoi* et *par qui* les données sont collectées, pas plus qu'il n'est possible de savoir combien de temps celles-ci seront conservées et qui pourra y accéder dans le futur. Par exemple, certaines informations sensibles, telles que l'état de santé ou les comportements de conduite, pourraient être utilisées à l'insu d'un utilisateur lorsque celui-ci tenterait de contracter une police d'assurance. De plus, dans le cadre des outils d'analyse de comportements suspects à priori [37], la présence d'erreurs introduites volontairement ou non serait directement et automatiquement dommageable pour les individus ciblés.

Ainsi, l'Internet des objets pose des problèmes de respect de la vie privée même pour ceux qui n'en utilisent pas les services, car la simple présence d'un individu dans l'environnement entraîne implicitement la collecte d'informations [7]. Couplé à l'identification unique des objets, cela rend possible la construction de profils extrêmement complets à propos des utilisateurs et des objets qu'ils possèdent ou transportent sur eux. Ces profils uniques peuvent ensuite être utilisés pour suivre les individus en temps réel et à très grande échelle, en se basant sur la détection des objets et sur les différents capteurs placés dans l'environnement [3]. Il faut avoir conscience que ce problème de collecte à l'insu des utilisateurs ne peut pas être résolu uniquement avec des mesures techniques. En effet, celles-ci doivent être accompagnées d'un cadre légal fort, notamment en ce qui concerne : (i) l'information des utilisateurs quant aux données collectées, (ii) les limitations légales relatives au stockage, à l'utilisation et à la redistribution de ces informations et, enfin, (iii) le droit des utilisateurs à faire rectifier ou supprimer tout ou partie des données collectées à leur sujet par un tiers [38].

Au-delà de la collecte à leur insu, les utilisateurs informés qui désirent bénéficier des services de l'Internet des objets doivent en outre gérer leurs données et le partage de celles-ci avec d'autres systèmes. Il est nécessaire pour les utilisateurs de pouvoir

définir la façon dont leurs objets collectent des informations à leur sujet (le *quoi*, le *où* et le *quand*) et à qui ces informations sont diffusées, ce de la manière la plus simple et compréhensible possible. Par ailleurs, selon le niveau de précision requis par les services avec lesquels l'utilisateur partage ses données, des mécanismes d'anonymisation (par exemple, la préagrégation des données de plusieurs objets de mêmes types possédés par plusieurs utilisateurs) ou d'obfuscation (réduction de la précision, introduction de fausses informations, etc.) devraient être proposés.

1.2 Vers un système distribué de gestion de flux pour l'Internet des objets

Étant donné les problématiques que nous venons d'évoquer pour la réalisation de l'Internet des objets, notre axe de travail principal consiste à étudier comment représenter et traiter l'ensemble de ces flux continus. On sait, en effet, que l'Internet des objets produit des informations qui dépendent essentiellement du temps, typiquement des échantillons et des évènements caractérisant une ressource. Aussi, cette question de la gestion des flux est primordiale et affecte directement les modèles de données de l'Internet des objets puisque, globalement, les techniques permettant d'accéder aux flux, de les traiter ou de les stocker sont spécifiques et diffèrent de celles utilisées pour manipuler des ensembles finis de données [29]. Par exemple, une divergence notable réside dans la façon dont les systèmes de traitement de flux stockent, typiquement, les informations et les logiques de transformations exprimées par les utilisateurs :

— Dans le cadre des ensembles finis de données, les informations sont généralement persistantes (bases de données, fichier, page Web, etc.) et les logiques de traitements sont généralement volatiles, c'est-à-dire conservées en mémoire jusqu'à ce que l'ensemble soit traité, après quoi elles sont « oubliées ».

— Dans le cadre des flux de données, qui sont infinis ou sans limites connues à priori – ce qui d'un point de vue théorique revient au même –, ce sont les logiques de traitement qui sont conservées indéfiniment (persistantes) tandis que les données sont typiquement traitées une seule fois puis « oubliées » (volatiles) [39].

Cette différence de conception a un impact fort sur la façon dont sont conçus les algorithmes de traitement de flux, qui doivent tenir compte de cette volatilité des données. En outre, stocker ou accéder aux flux de données implique (i) d'organiser les données de manière spécifique pour prendre en compte, notamment, les relations temporelles et (ii) de disposer des abstractions et des fonctions primitives conçues tout particulièrement pour la gestion des flux. Ces questions sont spécifiquement étudiées par la recherche sur les flux de données qui a permis l'émergence les *systèmes de gestion de flux de données* (*data stream management systems*, ou *DSMS*) [39–41]. Ces systèmes ont un rôle similaire aux systèmes de gestion de bases de données, mais permettent de gérer des flux (accès, stockage,

traitement, etc.) plutôt que des ensembles de données finis. Dans ce type d'approche, les traitements sont exprimés sous forme de requêtes qui sont ensuite exécutées soit (i) à partir des données collectées depuis les flux par un système centralisé, soit (ii) de manière distribuée au sein du réseau. La seconde approche est courante dans les réseaux de capteurs sans fil [42, 43] où l'on va souvent privilégier les interactions directes et le traitement distribué des données à l'intérieur du réseau (*in-network processing*), ce pour réduire les coûts en énergie induits par la communication sans fil [27, 44]. Par ailleurs, couplées au traitement continu où l'on ne stocke en mémoire que les données les plus récentes, les approches décentralisées ou distribuées sont théoriquement plus à même de supporter l'échelle de l'Internet des objets [4].

Cependant, seuls, les systèmes de gestion de flux de données ne suffisent pas pour gérer la complexité de l'Internet des objets. Au-delà des flux, la forte hétérogénéité fonctionnelle nécessite de promouvoir des mécanismes d'abstraction découplant les aspects fonctionnels et sémantiques des aspects techniques (capacités matérielles, interfaces de communication, protocoles supportés, etc.) et permettant, de fait, de manipuler les objets, leurs identités et leurs capacités sous une forme unifiée. De plus, réduire l'impact de l'hétérogénéité technique requiert de disposer d'une couche unificatrice destinée à masquer la complexité des mécanismes de découverte et d'accès aux objets hétérogènes. De ce fait, l'Internet des objets bénéficierait d'une plus grande interopérabilité et d'une plus grande flexibilité, simplifiant ainsi (i) les interactions entre les humains et les objets et (ii) les interactions entre objets eux-mêmes. En outre, conformément à ces abstractions et au contexte particulier de la gestion de flux, il est important de reconsidérer la problématique du déploiement des logiques de traitement exprimées par les utilisateurs. En effet, étant donné des tâches complexes produisant, transformant et consommant de multiples flux, quels sont les objets qui devraient prendre part à leurs exécutions ?

En conséquence, résumant tout ce que nous venons d'évoquer, notre thèse est la suivante :

> « *De par la nature continue des données (mesures physiques, évènements, etc.) et leur volume potentiel, il est important de considérer (i) les flux comme modèle de données de référence de l'Internet des objets et (ii) le traitement continu comme modèle de calcul privilégié pour transformer ces flux. En outre, étant donné les préoccupations croissantes en matière de consommation énergétique et de respect de la vie privée, il est préférable de laisser les objets agir au plus près des utilisateurs, si possible de manière autonome, au lieu de déléguer systématiquement l'ensemble des tâches à de grandes entités extérieures telles que le* cloud. *En considérant chaque objet comme un agent capable d'exécuter des tâches variées qui n'ont pas été définies à l'avance, l'Internet des objets pourra atteindre son plein potentiel et satisfaire aux exigences des cas d'utilisations mis en évidence dans la littérature.* »

Partant de là, l'ensemble de nos contributions est relatif au traitement de flux et son influence sur les mécanismes d'abstraction et d'unification de l'hétérogénéité de l'Internet

des objets à grande échelle. Cela passe tout d'abord par une compréhension de la nature des flux en tant que structure complexe et des opérations que l'on peut y appliquer, au travers d'un cadre formel conçu pour tenir compte des particularités de l'Internet des objets et de ses différents scénarios d'utilisation. Une fois ce cadre posé, nous pouvons l'utiliser pour reconsidérer plusieurs problèmes usuels, mais cette fois-ci du point de vue du traitement continu : accès aux données, déploiement des tâches de traitement continu et *in-network processing*.

Le premier problème que nous traitons porte sur les solutions destinées à réduire l'impact de l'hétérogénéité des objets, notamment l'*architecture orientée service* (*service-oriented architecture*, ou *SOA*) [45]. Il s'agit, en effet, d'une solution proposée dans la littérature pour atténuer les problèmes d'hétérogénéité de l'Internet des objets en le dotant des mécanismes de découverte et d'accès interopérables propres à cette architecture [46, 47]. Globalement, un service est une fonctionnalité autonome et atomique proposée par un *fournisseur de service*, ou *hôte*. Cette fonctionnalité peut être invoquée au travers d'un *protocole d'accès* qui masque l'ensemble des détails techniques liés à l'hôte et à l'exécution de la fonctionnalité. En outre, un agent (humain ou machine) peut découvrir simplement quels services sont disponibles en contactant des *annuaires* spécialisés qui référencent les différents hôtes et les services qu'ils proposent. Ainsi, dans le contexte spécifique de l'Internet des objets, il est possible de représenter les objets comme des fournisseurs de services (services de mesure, services d'action sur l'environnement, etc.), les annuaires devant être adaptés pour supporter l'échelle considérée [48]. Conformément à nos problématiques, nous avons étendu cette approche pour construire une architecture orientée service pour des fonctionnalités qui consomment et produisent des flux continus de données plutôt que des ensembles finis.

Le second problème que nous traitons porte sur le déploiement dynamique des tâches de traitement continu au sein de cette architecture. En effet, plusieurs scénarios propres à l'Internet des objets nécessitent de pouvoir déployer des traitements à tout moment sur un grand nombre d'objets ; par exemple dans le cadre du *sensing* partagé où les utilisateurs ouvrent leurs objets et leurs données à des tâches d'analyse collaborative. Dans cette optique, nous avons étudié le *problème du mapping de tâche*, qui consiste à déterminer quels sont les objets les plus adaptés aux caractéristiques d'une tâche donnée, conformément aux propriétés globales que l'on souhaite minimiser ou maximiser (consommation énergétique, capacité de traitement, fiabilité, temps de réponse, etc.).

Enfin, à partir de ces travaux théoriques, nous avons conçu *Dioptase*, un intergiciel orienté service pour le traitement des flux de données dans l'Internet des objets. *Dioptase* est une couche d'abstraction unifiée destinée à s'exécuter sur tous types d'objets allant des nouvelles générations de capteurs embarqués jusqu'aux infrastructures *cloud* en passant par les smartphones. Ainsi, chaque objet est une entité potentiellement autonome, capable de produire, de consommer et de traiter des flux sans qu'il soit nécessaire d'utiliser des intermédiaires, sauf dans des cas particuliers tels que les infrastructures de sécurité

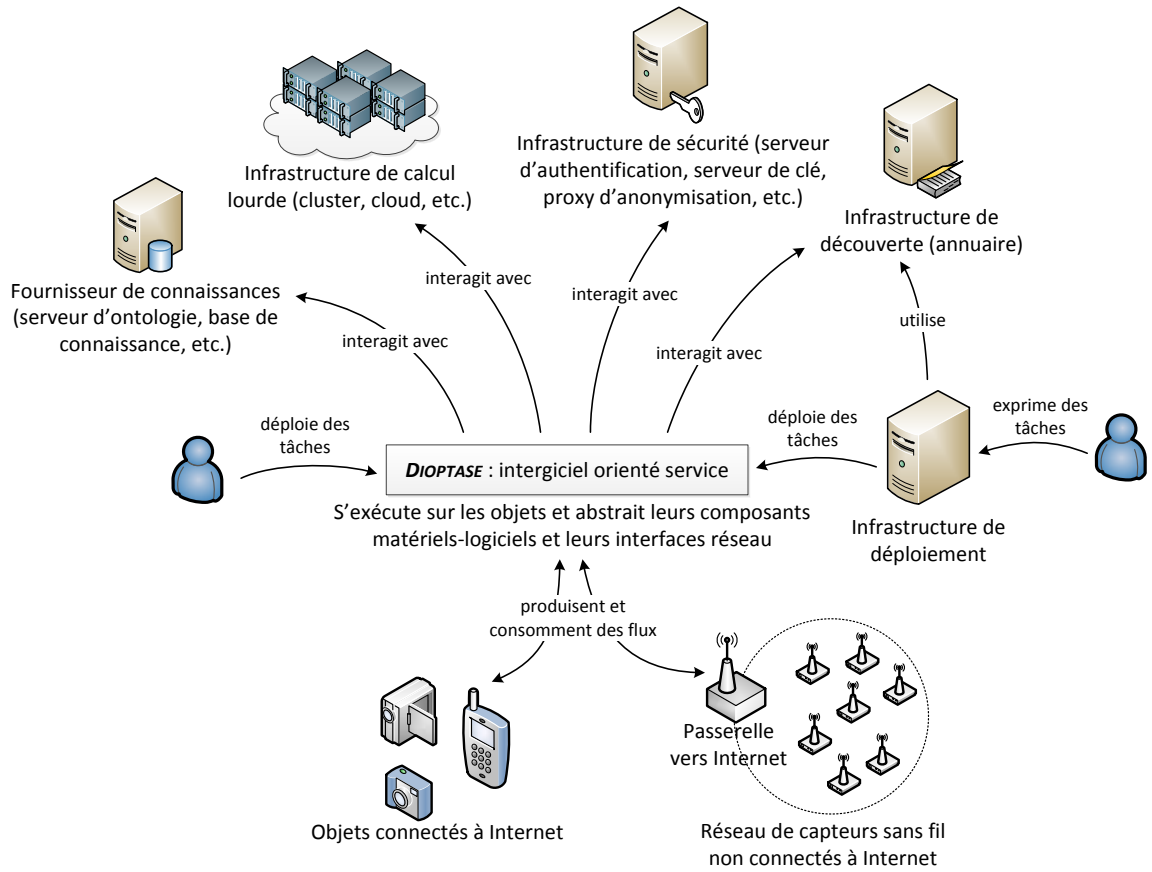


Figure 1.1 – *Dioptase*, un intergiciel de traitement de flux pour l'Internet des objets.

(pare-feu, proxy d'anonymisation, etc.) ou les passerelles entre réseaux de technologies différentes. En premier lieu, une fois déployé sur les objets, *Dioptase* fournit un cadre unifié pour représenter sous forme de flux les données produites par différentes sources, telles que des capteurs, des bases de données ou encore des informations saisies par les utilisateurs. En outre, *Dioptase* gère l'ensemble des métadonnées propres à l'objet : identité, ressources logicielles et matérielles, actionneurs, position, etc. En second lieu, *Dioptase* permet de transformer les objets en fournisseurs génériques de services de calcul et de stockage. Cela permet alors aux développeurs de construire des applications composées de tâches de traitement continu, puis de les déployer dynamiquement à tout moment, soit en passant par une infrastructure de déploiement (déploiement basé sur la résolution de *mapping* de tâches), soit en contactant directement les objets. Enfin, comme présenté sur la Figure 1.1, *Dioptase* fournit les briques logicielles pour s'interfacer avec les composants issus de la recherche sur les infrastructures pour l'Internet des objets, notamment en ce qui concerne la gestion de la sécurité et de la vie privée, la déportation du calcul intensif, la découverte des objets et la gestion de la sémantique [4, 7, 14, 15].

Conformément à ce que nous venons de décrire, ce manuscrit est organisé de la manière suivante :

— Le *Chapitre 2* est consacré à l'état de l'art de l'Internet des objets et à ses précurseurs que sont les réseaux de capteurs sans fil et les réseaux d'objets *RFID*. Nous

abordons les différentes visions qui sous-tendent l'Internet des objets, c'est-à-dire les différents objectifs et technologies clés proposés dans la littérature. En outre, nous présentons l'état de l'art spécifique au traitement de flux de données, aussi bien dans les réseaux de capteurs que dans le Web et le *cloud*.

— Le *Chapitre 3* introduit notre modèle de données (flux) et notre modèle de calcul (traitement continu). Nous y présentons une architecture de service continu, spécifiquement adaptée au traitement de flux, et un langage de traitement de flux associé.

— Le *Chapitre 4* porte sur l'exécution des services continus qui composent les applications orientées flux. Nous y présentons *Dioptase*, notre intergiciel de traitement continu pour l'Internet des objets, et son architecture.

— Le *Chapitre 5* est dédié au déploiement automatique des applications orientées flux au sein des réseaux d'objets. Nous y présentons *TGCA* (*Task Graph to Concrete Actions*), une formulation du problème de *mapping de tâche* dans le contexte de l'Internet des objets et une méthode de résolution appropriée.

— Le *Chapitre 6* aborde la problématique relative à l'intégration dans l'Internet des objets des systèmes déjà déployés dans l'environnement ; par exemple, les réseaux de capteurs sans fil. Nous apportons une solution à ce problème, sous la forme de *proxys opportunistes mobiles* capables de faire le lien entre les capteurs existants et l'Internet des objets.

— Le *Chapitre 7* conclut ce mémoire par un résumé des différentes contributions et leur mise en perspective vis-à-vis des problématiques de l'Internet des objets. En outre, nous y discutons des pistes de recherches futures à court et à long terme, étant donné que l'Internet des objets est un sujet vaste et un terrain de travail particulièrement riche.

La majeure partie des travaux présentés dans ce manuscrit a été publiée sous la forme d'articles scientifiques dans des journaux et des conférences à comité de lecture, ainsi que sous la forme de rapports techniques :

— Dans *D1.4 Final CHOReOS Architectural Style and its Relation with the CHOReOS Development Process and IDRE* [49], nous avons réalisé un état de l'art des systèmes de gestion de flux de données et identifié certaines fonctions primitives de la diffusion de flux, ainsi que les relations qui peuvent exister entre ces primitives et celles d'autres méthodes de communication.

— Dans *From Task Graphs to Concrete Actions : A New Task Mapping Algorithm for the Future Internet of Things* [50], nous avons présenté nos travaux sur le problème de *mapping* de tâche dans l'Internet des objets, tout particulièrement sous le prisme du traitement continu de flux de données, et proposé un algorithme de résolution pouvant être déployé à l'intérieur du réseau.

— Dans *Dioptase : A Distributed Data Streaming Middleware for the Future Web of Things* [51], nous avons décrit les recherches autour de *Dioptase*, notre intergiciel pour le traitement distribué de flux dans l'Internet des objets : ses objectifs, son architecture et les expérimentations que nous avons réalisés à son sujet.

— Dans *Spinel : An Opportunistic Proxy for Connecting Sensors to the Internet of Things* [52] (en cours de soumission), nous avons travaillé sur l'ouverture à Internet des réseaux de capteurs déjà déployés dans l'environnement, en proposant une solution basée sur des proxys mobiles découvrant et présentant les capteurs sur Internet de manière opportuniste.

Chapitre 2

Internet des objets et traitement de flux : état de l'art

Les concepts prometteurs de l'Internet des objets reposent sur deux décennies de progrès technologiques et de travaux dans des domaines divers. Dans ce chapitre, nous revenons sur les précurseurs de l'Internet des objets que sont les réseaux de capteurs sans fil et les réseaux d'objets *RFID*, ainsi que sur leur intégration au réseau Internet et au Web. Dans un second temps, nous abordons la problématique du traitement des flux de données et posons les concepts spécifiques à la littérature du traitement continu pour les réseaux de capteurs et pour les systèmes *cloud*.

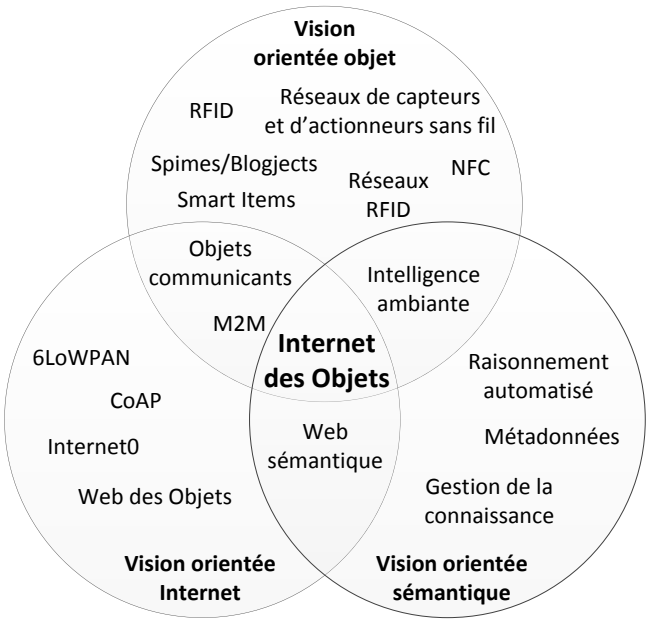


Figure 2.1 – L'Internet des objets, émergeant de différentes visions.

2.1 Visions de l'Internet des objets

L'Internet des objets émerge des travaux menés dans plusieurs domaines de recherche dont les différentes « visions » convergent vers un même but. Dans leur état de l'art, *Atzori et coll.* identifient trois visions majeures : une vision *orientée objet*[1], une vision *orientée Internet* et une vision *orientée sémantique* [7]. Comme montré sur la Figure 2.1, chaque vision possède ses concepts et ses technologies clés.

La vision *orientée objet* met l'accent sur les objets physiques et les systèmes qui y sont embarqués, avec notamment les problématiques suivantes :

— identifier les objets de manière unique et, de fait, les doter d'une identité propre ;

— permettre aux objets d'acquérir des informations sur leur environnement en utilisant des capteurs, pour ensuite échanger et traiter ces informations afin d'influer sur cet environnement au travers d'actionneurs.

Cette vision est principalement issue des travaux sur les réseaux d'objets identifiés par radiofréquence (*RFID*), la communication en champ proche (*near field communication*, ou *NFC*) et les réseaux de capteurs et d'actionneurs sans fil (*wireless sensor and actuator network*, ou *WSAN*) [7]. L'Internet des objets y est vu comme le concept émergeant de la mise en réseau des différents objets dans le but de faciliter les interactions entre les êtres humains et les objets, ainsi que les interactions entre objets eux-mêmes.

La vision *orientée Internet* suggère que cette interconnexion devrait se faire spécifiquement au travers du réseau Internet, considérant que celui-ci est déjà un « réseau de réseaux » qui connecte un très grand nombre de machines au moyen de protocoles

1. Il s'agit ici de la traduction du terme *Thing-oriented* où « objet » désigne des objets physiques, à ne pas confondre avec le paradigme de programmation orientée objet (*object-oriented programming*).

standards conçus pour de nombreux usages. Ainsi, une seule couche réseau, au sens OSI [53] du terme, est utilisée pour connecter tous types d'objets, simplifiant de fait la fabrication et le déploiement de ces derniers, tout en assurant une certaine interopérabilité [54]. Cette vision s'attache donc à étudier comment le protocole *IP* (*Internet Protocol*) peut être adapté pour des systèmes embarqués caractérisés par de faibles ressources matérielles, notamment au travers de nouveaux standards. Par extension, si l'on connecte les objets physiques au réseau Internet, il devient possible de les intégrer aussi au Web pour bénéficier de ses avantages (Web des objets). En effet, le Web connecte aujourd'hui avec succès un très grand nombre de systèmes très hétérogènes [55] que les êtres humains utilisent au quotidien, le Web étant l'application d'Internet la plus utilisée par le grand public à l'heure actuelle. Ainsi, utiliser le Web comme support à l'Internet des objets permettrait : (i) aux machines de directement manipuler les ressources déjà disponibles dans le très vaste monde virtuel et (ii) aux humains d'interagir avec ces machines de façon naturelle [55].

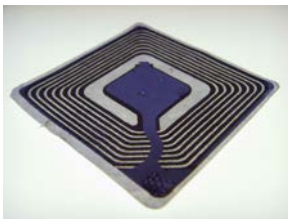
Enfin, la vision *orientée sémantique* se détache des problématiques techniques propres aux objets physiques et aux réseaux permettant de les faire communiquer, pour se concentrer sur la représentation, l'organisation et le stockage des données relatives à l'Internet des objets, notamment :

— les identités de chaque objet physique (caractéristiques, états passés et présents, etc.) et les relations qui peuvent unir ces objets entre eux ;

— les flux d'informations acquis par ces objets sur leur environnement, par exemple au moyen de capteurs.

En effet, décrire et structurer ces informations ainsi que les liens qui les unissent, au moyen de formats manipulables par les machines, est un problème fondamental pour permettre à ces dernières de collaborer de manière autonome. Dans ce type d'approche, la sémantique (c.-à-d. le sens) des données est distincte des données elles-mêmes [3] et est représentée sous forme de métadonnées. Ces métadonnées permettent alors aux machines d'analyser plus efficacement les grands volumes d'informations produits par l'Internet des objets, au travers des mécanismes usuels utilisés pour la classification et le regroupement de données ainsi que la recherche de schémas ou de corrélations utiles. En outre, les machines peuvent raisonner automatiquement à partir de ces informations et adopter des comportements plus intelligents en fonction du contexte d'utilisation et de l'environnement [5]. Enfin, la gestion de la sémantique au sein de l'Internet des objets permet aux êtres humains d'interagir plus facilement avec les machines, par exemple en utilisant le langage naturel, et de recevoir des informations plus pertinentes [56].

La vision orientée objet et la vision orientée Internet partagent l'ambition commune d'interconnecter des systèmes divers au sein d'un réseau global. Cette problématique concerne essentiellement deux catégories d'objets, qui sont les objets *RFID* (passifs) et les réseaux de capteurs (actifs) qui permettent d'obtenir des informations sur le monde physique et d'agir sur celui-ci en conséquence. En effet, les autres objets du quotidien qui



(a) Étiquette électronique.

(b) Marqueur sous-cutané.

Figure 2.2 – Exemples de puces *RFID*.

permettent déjà d'interagir avec le monde virtuel (ordinateurs, smartphones, tablettes, etc.) sont déjà connectés au réseau Internet et communiquent à grande échelle. Enfin, les problématiques de la vision orientée sémantique sont orthogonales aux problématiques du réseau global : quelles que soient les solutions choisies pour la communication des objets, les problématiques de représentation et de raisonnement demeurent.

Comme nous l'avons présenté dans l'introduction de ce mémoire, l'Internet des objets est caractérisé par les problèmes liés à sa grande échelle, à l'hétérogénéité des objets et à la nature des données issues du monde physique. Chaque vision apporte ses propres solutions à un ou plusieurs de ces problèmes conformément au cadre qui lui est propre. Aussi, la réalisation d'une couche intergicielle pour la gestion de l'Internet des objets passe avant tout par l'étude de la diversité des objets physiques qui forment l'environnement. En effet, à l'origine orienté vers les réseaux *RFID*, l'Internet des objets a été enrichi par les travaux sur les réseaux de capteurs sans fil et bénéficie aujourd'hui des nouveaux standards pour l'accès et le contrôle des objets au travers du Web (Web des objets). Une fois le cadre posé quant à l'environnement physique, nous revenons sur la problématique de l'échelle et les différentes solutions proposées dans la littérature. Enfin, nous étudions la littérature propre au traitement continu des flux de données, aussi bien dans le cadre général que dans le cadre spécifique des données dépendantes du temps produites par les réseaux de capteurs. Tous ces travaux nécessitent aujourd'hui d'être intégrés les uns avec les autres, notamment le Web des objets et le traitement de flux, cette intégration étant l'un des objectifs de nos travaux.

2.1.1 Étiquettes *RFID*

Le terme Internet des objets a été initialement utilisé pour décrire des objets identifiés de manière unique au moyen d'une puce *RFID*, aussi appelée étiquette *RFID* (*tag RFID*), pouvant être lue à distance au moyen d'un lecteur idoine. Possédant une identité virtuelle associée à leurs identifiants, les objets ainsi marqués peuvent être suivis dans l'espace en fonction des lecteurs à proximité et leurs positions peuvent être recoupées avec des informations acquises en temps réel depuis l'environnement [2, 33, 57].

En pratique, les puces *RFID* sont utilisées dans de nombreux contextes et, tout particulièrement, comme support aux applications logistiques en temps réel (gestion d'entrepôts

et de stocks, suivi d'objets pendant leur transport, etc.) et aux technologies d'identification (contrôle d'accès, lien entre les personnes et les systèmes d'informations, etc.). Les étiquettes *RFID* les plus répandues sont dites *passives* et se composent uniquement d'une antenne et d'une puce électronique capable de répondre aux requêtes des lecteurs (récupération de l'identifiant). Ne possédant pas de batteries, les puces *RFID* passives extraient directement leur énergie du champ électromagnétique émis par les lecteurs *RFID* lorsque ceux-ci tentent de les contacter. Du fait de leur capacité à être lue à distance, de leur taille minime et de leur coût de production négligeable, estimé à quelques centimes par unité [33], ces dispositifs sont utilisés massivement, par exemple comme évolution des codes-barres pour :

— les biens de consommation (étiquettes électroniques) ;

— les animaux (marqueur sous-cutané) ;

— les systèmes d'identification, de suivi et de paiement, sous la forme de cartes sans contact : passeports, badges d'accès, cartes de crédit ou de transport, etc.

Il existe aussi des puces *RFID* dites *actives*, équipées d'une batterie interne et dotées de ressources matérielles plus importantes. De ce fait, celles-ci peuvent stocker un plus grand nombre d'informations et possèdent des capacités de communication accrues leur permettant d'interagir avec d'autres appareils actifs ou non. En outre, la distance d'émission se trouve significativement accrue par la présence d'une source d'énergie, pouvant aller jusqu'à rendre possible une lecture par satellite [33]. Ces dispositifs actifs peuvent aussi embarquer des capteurs leur permettant de collecter des informations sur leur environnement. On trouve, par exemple, des puces *RFID* actives capables (i) de mesurer des paramètres physiques tels que la température, l'humidité et la luminosité, (ii) de déterminer leur position au moyen d'un GPS, ou encore (iii) de détecter des accélérations ou des rotations [58]. Le champ d'application de tels systèmes est très large, les capteurs étant directement intégrés à la puce et donc directement liés à l'objet sur lequel celle-ci est placée. Par exemple, une puce *RFID* intégrée dans un emballage alimentaire peut incorporer un capteur de température ou un dispositif pour mesurer la prolifération des micro-organismes [59]. Cependant, ce type de système est significativement moins répandu que les dispositifs passifs, du fait de leur coût unitaire nettement plus élevé et de la durée de vie limitée de leur batterie. Par ailleurs, un effort de recherche conséquent est mené pour permettre aux puces *RFID* passives d'embarquer des capteurs sans alimentation ou capables de fonctionner uniquement avec l'énergie extraite du champ électromagnétique généré par le lecteur [59, 60].

Enfin, certaines puces sont dites *semi-passives*. Tout comme les puces actives, celles-ci sont équipées d'une source d'énergie (batterie, capteur solaire, etc.), mais ne peuvent pas initier de communication à l'instar des puces passives. La source d'énergie est ici utilisée pour pouvoir effectuer des calculs plus complexes lorsque la puce est lue, ou pour alimenter des capteurs embarqués dont les mesures seront lues en même temps que la puce.

Classe 0	Puces passives en lecture seule possédant un identifiant stocké en dur à la construction (*factory-programmed*).
Classe 1	Puces passives vierges pouvant être écrites une seule fois (*user-programmed*).
Classe 2	Puces passives vierges pouvant être écrites plusieurs fois et possédant des mécanismes supplémentaires ; par exemple, le chiffrement du contenu.
Classe 3	Puces semi-passives intégrant une source d'énergie (batterie, capteurs solaires, etc.) et pouvant effectuer des actions sans être alimentées par un lecteur *RFID* ; par exemple, capter des informations sur l'environnement.
Classe 4	Puces actives capables d'initier des communications avec d'autres puces de classe 4 (ad hoc) ou des lecteurs *RFID*.
Classe 5	Puces actives capables de communiquer avec toutes les autres classes ; par exemple, un lecteur *RFID* est de classe 5.

Table 2.1 – Les différentes classes de puces *RFID*.

Les différents types de puces *RFID* sont regroupés en classes en fonction de leurs capacités [61], ces classes étant présentées dans la Table 2.1. Globalement, toutes les puces possèdent un ensemble de fonctionnalités communes, comme la possibilité d'être lues par un lecteur. En outre, depuis la deuxième génération de puces *RFID* introduite en 2004 (*EPC Class 1 Generation 2*) [62], les étiquettes fournissent un mécanisme permettant à un lecteur de les désactiver (*kill command*) au moyen d'un message spécifique. Le comportement d'une puce à la réception du message dépend de sa configuration ; celle-ci peut soit effacer tout ou partie de sa mémoire, soit ne plus répondre ultérieurement aux lecteurs, soit ignorer la commande de désactivation. Par ailleurs, une puce peut supporter des mécanismes d'authentification par mot de passe (32 bits), dans le but de limiter la lecture et la désactivation à des entités autorisées.

La technologie d'identification la plus répandue pour les étiquettes *RFID* est celle des *codes de produit électroniques* (*Electronic Product Code* [19], ou *EPC*), qui sont représentés sous la forme d'un *URI* pouvant supporter plusieurs formats d'identification universelle unique. À partir de cet *URI*, le standard *EPC* définit comment produire un identifiant binaire compact (64, 96 ou 125 bits) qui peut ensuite être stocké dans une étiquette *RFID* ou encore exprimé dans un format de codes-barres classiques. En pratique, le standard ne spécifie pas l'utilisation d'une technologie d'identifiant unique, mais permet, au contraire, d'utiliser différents standards en fonction du domaine.

2.1.2 Réseaux de capteurs sans fil

Outre les réseaux *RFID*, où les objets étiquetés sont dépendants d'autres objets (les lecteurs) pour pouvoir être détectés et connectés au réseau global, les systèmes embarqués communicants sont une part très importante de l'Internet des objets.

Figure 2.3 – Exemples de capteurs sans fil.

Les réseaux de capteurs sans fil, ou *wireless sensor network* (*WSN*) en anglais, sont des systèmes distribués composés d'un grand nombre d'appareils capables d'acquérir des informations sur leur environnement au moyen de capteurs embarqués : température, luminosité, vibrations, présence, etc. Typiquement, ces dispositifs sont équipés d'une batterie et communiquent au moyen de liaisons sans fil bas débit et à courte portée, ce qui les rend particulièrement autonomes. En effet, dotés de capacités matérielles réduites et, de fait, peu coûteux, ils sont destinés à être déployés en masse dans l'environnement et à produire des mesures en continu, ce avec un minimum d'intervention humaine. Versatiles, les réseaux de capteurs sans fil ont été utilisés pour de nombreuses applications [27], par exemple dans le domaine médical [63], les maisons et villes intelligentes [64, 65], ou encore l'étude des populations animales [66].

De manière générale, les réseaux de capteurs sans fil s'articulent autour de deux grandes familles d'appareils qui interagissent entre eux :

— Les *nœuds* (ou *motes*), qui sont les dispositifs les plus répandus dans le réseau. Ils sont, le plus souvent, dotés d'un processeur peu puissant, d'une mémoire limitée et d'une interface de communication sans fil. Ce sont eux qui effectuent les mesures sur le terrain, grâce aux capteurs qu'ils embarquent.

— Les *bases*, qui sont des dispositifs beaucoup plus performants, mais qui ne possèdent pas de capteurs. Elles sont généralement en nombre limité dans le réseau, et sont utilisées (i) comme points de collecte centralisée recevant les mesures acquises par les nœuds, (ii) comme intermédiaires entre deux réseaux de capteurs, ou avec un autre réseau (Internet, notamment) ou (iii) comme points d'interaction avec les utilisateurs finaux.

En fonction des usages et des cas d'utilisation de chaque réseaux de capteurs, les différents appareils sont dotés de capacités et de comportements spécifiques [27], notamment en matière de communication, d'interaction et de mobilité. La communication, tout particulièrement, est cruciale dans ce type de systèmes, étant donné l'absence d'infrastructure dédiée à l'échange d'informations. En effet, les nœuds communiquent entre eux et avec la base au travers de leurs liaisons sans fil sans requérir un point d'accès central. Cette communication a un coût énergétique élevé, tout particulièrement en ce

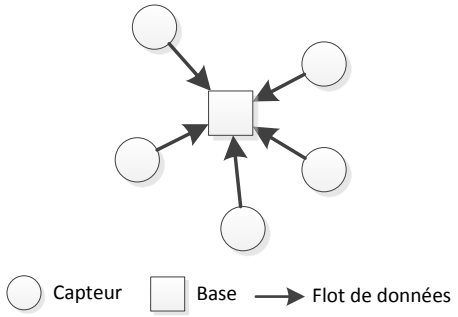


Figure 2.4 – Collecte centralisée (en étoile).

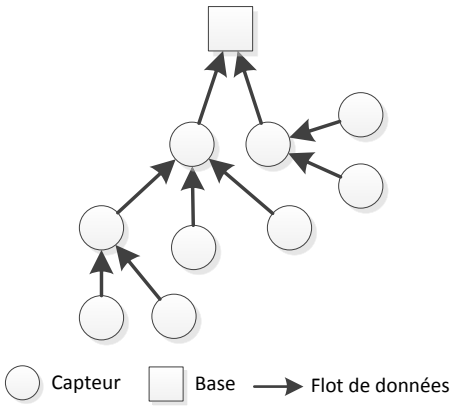


Figure 2.5 – Arbre de collecte.

qui concerne l'émission de données : un seul bit envoyé peut parfois consommer autant d'énergie que l'exécution d'un millier d'instructions par le processeur [67]. Pour réduire la consommation énergétique, de nombreuses technologies de nœuds n'autorisent les communications que pendant des intervalles périodiques (*time slots*), l'interface réseau étant désactivée le reste du temps.

Lorsque les nœuds échangent des données, le cas le plus simple est celui de la communication *mono-saut*, où chaque appareil échange des informations uniquement avec les dispositifs qui sont suffisamment proches pour communiquer. Cependant, la portée de ces liaisons est plutôt réduite [2], ce pour minimiser la consommation énergétique. Aussi, les réseaux de capteurs sans fil de grandes tailles privilégient la communication *multi-sauts*, où chaque nœud est potentiellement capable d'agir comme un intermédiaire (rôle de routage) entre d'autres nœuds. Ces réseaux, dits *réseaux ad hoc*, s'auto-organisent pour construire les routes par lesquelles les messages sont amenés à transiter.

Qu'il s'agisse de communication mono-saut ou multi-sauts, on trouve deux cas d'utilisation majeurs pour les réseaux de capteurs sans fil, qui consistent à utiliser le réseau pour effectuer soit (i) des tâches de mesure seule, auquel cas les nœuds embarquent uniquement des capteurs, soit (ii) des tâches de mesure et d'influence sur l'environnement [70]. Dans ce second scénario, les nœuds intègrent des mécanismes qui leur permettent d'interagir avec les utilisateurs (par exemple, une alerte lumineuse ou sonore) ou d'effectuer des actions qui modifient les caractéristiques de l'environnement (par exemple, un climatiseur ou un dispositif d'ouverture-fermeture de porte). Ces mécanismes sont appelés *actionneurs* et l'on parle alors d'un réseau de capteurs et d'actionneurs sans fil, ou *wireless sensor and actuator network* (*WSAN*). Ces réseaux sont typiquement employés pour mettre en œuvre des scénarios où les nœuds équipés d'actionneurs réagissent aux mesures produites par les nœuds équipés de capteurs.

2. La portée des liaisons sans fil pour réseaux de capteurs est généralement inférieure à 50 mètres. Globalement, pour réduire la consommation énergétique associée à la communication, les liaisons sont soit (i) à courte portée et à débit moyen comme, par exemple, *IEEE 802.15.4* [68], soit (ii) à longue portée et à débit très faible comme, par exemple, le réseau *Sigfox* [69].

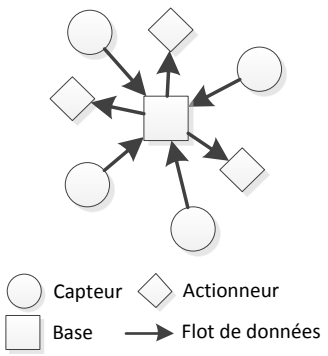


Figure 2.6 – Collecte et contrôle centralisés.

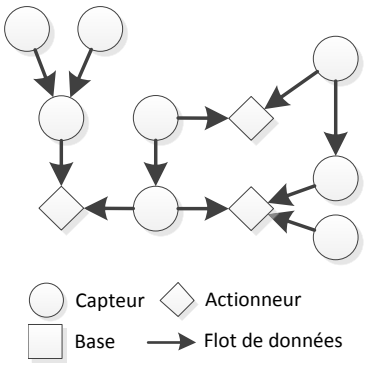


Figure 2.7 – Collecte et contrôle distribués.

Les scénarios de mesure seule utilisent souvent une approche centralisée, où les nœuds envoient l'ensemble de leurs mesures à la base. Cette dernière les stocke, généralement dans une base de données, et permet aux utilisateurs de récupérer et de traiter ces informations à posteriori [27]. Ici, le mode mono-saut peut être utilisé si les capteurs sont suffisamment proches de la base. Toutefois, pour les réseaux de capteurs plus étendus, il devient nécessaire d'employer une communication multi-sauts. Pour ce faire, une méthode courante consiste à faire appel à un *arbre de collecte*. Dans cette approche, illustrée par la Figure 2.5, les nœuds transmettent leurs mesures à des *nœuds parents* qui vont router ces données vers leurs propres parents, et ainsi de suite jusqu'à atteindre la base [42]. Selon les approches, les nœuds sont soit (i) passifs et attendent qu'un parent les contacte, soit (ii) actifs et prennent l'initiative de découvrir et de s'associer directement aux parents les plus proches.

En ce qui concerne les scénarios impliquant aussi bien des capteurs que des actionneurs, il est nécessaire de gérer la logique de contrôle des actionneurs [71]. Une solution simple consiste à héberger cette logique de manière centralisée au niveau de la base, qui joue alors un rôle de coordinateur. En effet, puisque cette dernière collecte les mesures issues des capteurs, elle possède toutes les informations nécessaires pour contrôler les actionneurs, comme montré sur la Figure 2.6. En pratique, cette solution présente des inconvénients lorsque le nombre de capteurs et d'actionneurs augmente ; d'une part, la base est rapidement surchargée et, d'autre part, le coût énergétique global et la latence sont accrus par la circulation des messages [71]. Pour résoudre ce problème, il a été proposé de distribuer la logique de contrôle des actionneurs directement dans le réseau. Cette approche modifie la façon dont les données circulent dans le réseau (typiquement multi-sauts), les capteurs produisant des informations qui sont consommées directement par les actionneurs, comme illustré sur la Figure 2.7. Enfin, lorsque les réseaux de capteurs deviennent très grands, il peut être intéressant de tirer parti de plusieurs bases réparties géographiquement. Il s'agit d'une approche typiquement décentralisée, où chaque base gère un groupe de nœuds et interagit avec les autres bases. Ici, la communication entre

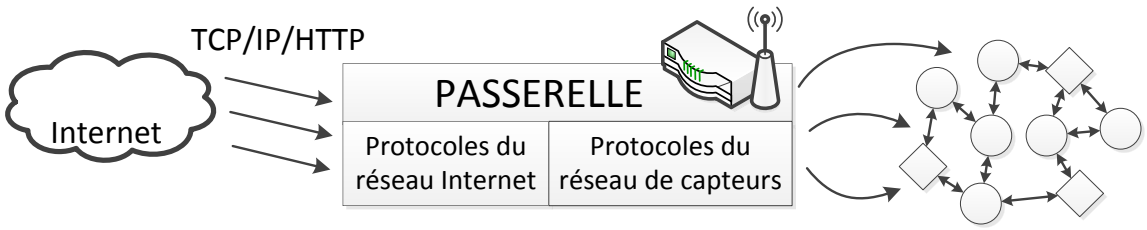


Figure 2.8 – Ouverture d'un réseau de capteurs à Internet au travers d'une passerelle.

les bases peut être assurée par une infrastructure de communication dédiée, par exemple le réseau Internet [72].

2.1.3 Ouverture des objets à Internet

Les capteurs et les actionneurs, tout comme les objets identifiés par *RFID*, sont des composants majeurs de l'Internet des objets. Par ailleurs, l'Internet des objets présente des caractéristiques communes avec les réseaux de capteurs sans fil, étant lui même un réseau composé, entre autres, de capteurs et d'actionneurs sans fil hétérogènes, potentiellement mobiles et limités en énergie. Toutefois, la principale problématique concerne l'échelle de l'Internet des objets par rapport à celle des réseaux de capteurs sans fil. En effet, en comparaison de l'Internet des objets, les réseaux de capteurs sans fil ont tendance à être isolés les uns des autres, ce pour deux raisons historiques :

— ils sont dédiés à l'accomplissement de tâches et de scénarios bien définis et bien délimités, pour le compte d'entités précises [27], les informations collectées restant confinées dans chaque réseau ;

— ils utilisent des technologies hétérogènes et propriétaires [26], aussi bien au niveau matériel (architecture et composants, interfaces de communication) que logiciel (interfaces de programmation, protocoles), ce qui a pour effet de limiter les facultés des réseaux à échanger entre eux.

Lorsque nécessaire, l'ouverture des réseaux de capteurs dépend de composants logiciels et matériels spécifiques capables d'effectuer la traduction entre deux réseaux propriétaires ou avec le réseau Internet (voir Figure 2.8) : les passerelles (*gateway*) et les proxys [3] [73], qui sont typiquement implémentés dans les bases. Les utilisateurs qui désirent accéder aux informations ou transmettre des ordres à un réseau de capteurs donné doivent tout d'abord se connecter ou s'identifier auprès de la passerelle. Ce processus est difficilement automatisable, étant donné que chaque passerelle est spécifique à une ou plusieurs technologies de réseaux de capteurs [26]. Aussi, pour résoudre ces problèmes,

3. Dans le vocabulaire des réseaux de capteurs sans fil, la différence entre une passerelle et un proxy n'est pas parfaitement définie, les deux termes étant utilisés de manière interchangeable [73]. Dans le cadre du réseau Internet, un proxy est une passerelle dotée de capacités supplémentaires (filtrage, compression, mise en cache, etc.)

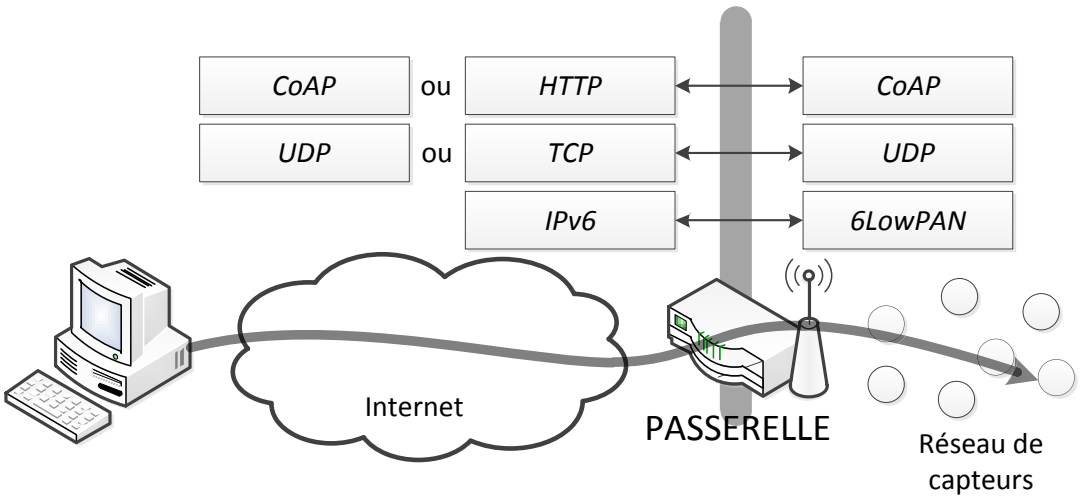


Figure 2.9 – Traduction entre la pile *TCP/IP* et la pile *6LoWPAN*/*CoAP*.

un effort de standardisation a été amorcé par l'*Internet Engineering Task Force* (*IETF*) [4] pour rendre possible l'intégration des systèmes embarqués au réseau Internet. Ce travail s'effectue aussi bien au niveau de la couche réseau, notamment avec la transmission de paquets *IPv6* dans des réseaux à faible consommation énergétique et fort taux de pertes (*low-power and lossy networks*, ou *LLN*) [74–76], qu'au niveau de la couche application, notamment pour l'invocation de services sur des appareils peu puissants [77].

Concrètement, l'objectif de l'*IETF* consiste à produire une pile standard de protocoles adaptés aux systèmes embarqués (ressources limitées, réseaux sans fil, etc.) [78] et à spécifier comment traduire ces protocoles vers la pile standard du réseau Internet, comme le montre la Figure 2.9 [26]. Ainsi, les passerelles ne sont plus liées à un réseau de capteurs propriétaire et peuvent être produites par n'importe quel fabricant, voire intégrées directement au sein des routeurs du réseau Internet. Au final, l'interopérabilité des systèmes embarqués est significativement accrue, ces derniers étant accessibles depuis l'extérieur sans qu'à aucun moment il ne soit nécessaire d'introduire des composants propriétaires.

La pile de protocoles introduite par l'*IETF* pour les réseaux de capteurs est équivalente à la pile *TCP/IP* du réseau Internet, ce pour faciliter la traduction entre les deux. Les différents protocoles sont les suivants : *IEEE 802.15.4* [68] pour la couche physique-liaison, *6LoWPAN* [76] pour la couche réseau et *UDP* pour la couche transport, ce dernier étant déjà utilisé sur le réseau Internet conjointement à *TCP*. En outre, l'*IETF* introduit un protocole d'invocation de services optimisé pour les systèmes restreints en ressources, *CoAP*. Ce dernier est notamment prévu pour servir de base au Web des objets, une extension de l'Internet des objets que nous aborderons un peu plus loin. Au final, les différents protocoles conçus par l'*IETF* pour les *LLN* font le lien entre les visions orientées Internet et orientées objets que nous avons présentées en introduction de ce chapitre.

4. L'*IETF* est un groupe international dédié au développement des standards concernant le réseau Internet, et plus particulièrement ceux propres à la pile de protocoles *TCP/IP*.
https://www.ietf.org (accédé le 01/08/2014)

Le standard *IEEE 802.15.4*, maintenu par l'*IEEE 802.15 WPAN Task Group 4*[5], spécifie des protocoles pour la couche physique et la couche liaison des réseaux sans fil caractérisés par un débit réduit, une portée restreinte et une faible empreinte énergétique. Par la suite, le standard a fait l'objet de plusieurs amendements destinés à améliorer sa fiabilité, à réduire sa consommation énergétique dans le cas des communications multi-sauts [78], ou à permettre son utilisation dans les *medical body area network*[6] [79]. De manière générale, les réseaux *IEEE 802.15.4* (*wireless personal area network*, ou *WPAN*) se composent de deux types d'appareils : les *full-function device* (*FFD*), qui sont capables de jouer le rôle de coordinateur au sein du réseau, et les *reduced-function device* (*RFD*) qui sont très limités en ressources et ne peuvent réaliser que des tâches très simples. Par ailleurs, le comportement de ces derniers consiste à s'associer avec un *FFD* et à ne communiquer qu'au travers de celui-ci. Au sein du réseau, les *FFD* s'organisent entre eux selon une topologie définie par le coordinateur principal (ou *PAN coordinator*), c.-à-d. le *FFD* ayant construit le *WPAN* en générant un identifiant spécifique pour celui-ci (*PAN ID*). En pratique, le standard propose plusieurs modes spécifiant quelle bande de fréquence et quel type de modulation doivent être utilisés, ce qui conditionne les débits théoriques (de 20 Kbps à 1 Mbps) ainsi que la portée (de 10 à 100 mètres) pour chaque mode. Plusieurs modes sont cependant optionnels et rarement supportés en pratique, les modes requis supportant un débit compris en 20 Kbps et 250 Kbps (mode le plus courant) [26].

Comme le standard *IEEE 802.15.4* ne spécifie que les mécanismes de couches physique et liaison, celui-ci sert de base pour des protocoles de couches supérieures, tels que *ZigBee* [80] ou, dans le cas qui nous intéresse, *6LoWPAN*. Ce protocole résulte d'une volonté d'ouvrir les réseaux de capteurs sur le réseau Internet, en dotant les nœuds d'une adresse *IPv6* [74]. L'espace d'adressage *IPv6*, avec ses 2^{128} adresses possibles, présente en effet l'avantage d'être suffisamment grand par rapport à l'échelle estimée de l'Internet des objets [7]. Toutefois, l'utilisation du protocole *IPv6* tel quel n'est pas envisageable dans les réseaux bas débit. Par exemple, *IEEE 802.15.4* fonctionne avec des trames limitées à 127 octets (entêtes compris), avec une charge utile comprise entre 106 et 118 octets. Comme un paquet *IPv6* contient au minimum 40 octets d'entêtes, cela reviendrait à consacrer environ 35% de la charge utile au seul adressage [74]. En outre, la charge utile minimale d'un paquet *IPv6* est de 1280 octets ; la fragmentation n'étant plus supportée de manière automatique en *IPv6*, les réseaux ne supportant pas cette taille de paquet doivent supporter eux-mêmes les mécanismes de fragmentation [18]. Aussi, *6LoWPAN* est une version simplifiée et allégée d'*IPv6*, spécialement conçue pour supporter les réseaux *IEEE 802.15.4*. Pour ce faire, *6LoWPAN* propose, en premier lieu, des mécanismes de compression d'entêtes qui réutilisent l'adressage *IEEE 802.15.4* et qui se spécialisent selon les situations (adresses locales, adresses globales, unicast, multicast, couche supérieure

5. *http://www.ieee802.org/15/pub/TG4.html* (accédé le 07/07/2014).

6. Un *Medical Body Area Network* (*MBAN*) est un réseau sans fil composé d'appareils disposés sur, autour ou dans le corps humain, dont le but est d'acquérir des informations sur un patient.

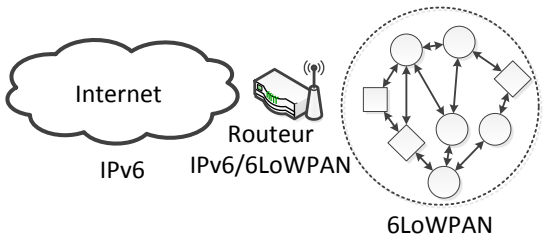


Figure 2.10 – Traduction *IPv6-6LoWPAN*.

spécifique, etc.) [26, 81]. Dans un second temps, *6LoWPAN* introduit les mécanismes nécessaires (i) à la fragmentation des paquets *IPv6* lorsque ceux-ci dépassent 106 octets et (ii) au routage dans les topologies pair-à-pair *IEEE 802.15.4* [76]. De nombreux autres mécanismes sont prévus par le *6LoWPAN*, notamment pour l'autoconfiguration des nœuds (négociation d'adresse) et leurs découvertes mutuelles.

En pratique, ce sont les routeurs qui convertissent les paquets entre les réseaux *IPv6* et les réseaux *6LoWPAN*, comme montré sur la Figure 2.10. Ils prennent notamment en charge la compression des adresses et leur traduction, ainsi que la fragmentation des paquets en provenance de l'extérieur. Ici, c'est donc le routeur qui tient le rôle de passerelle, la différence principale entre les deux étant la disparition des composants propriétaires, les routeurs étant des composants standards du réseau Internet. À l'intérieur même des réseaux *6LoWPAN*, le routage est assuré au moyen d'un protocole de routage appelé *RPL* [76], spécifiquement conçu pour les *LLN* de type *IEEE 802.15.4*. Ainsi, grâce à ces différents standards, les systèmes embarqués limités en ressources peuvent être considérés de la même façon que n'importe quel autre appareil connecté à Internet, et notamment être exploités au moyen de services Web.

Ces nouveaux standards, formant une pile *IP* complète pour les systèmes embarqués permet au final d'ouvrir les objets à Internet à moindre coût et ouvre la voie au Web des objets, l'étape suivante de l'Internet des objets consistant à ouvrir les objets sur le Web de façon à les manipuler comme n'importe quelle autre ressource du Web (pages, services, etc.).

2.1.3.1 Le Web des objets

Le Web est une application du réseau Internet qui permet, à l'origine, d'accéder à des pages Web reliées par des liens hypertextes. Par la suite, le Web a évolué jusqu'à devenir un ensemble de technologies permettant de représenter des ressources identifiées par des adresses uniques (*URI*), ces ressources pouvant être des pages Web, mais aussi des fichiers, des flux ou des services. À l'heure actuelle, le Web interconnecte un très grand nombre d'appareils fortement hétérogènes et permet aux utilisateurs d'y accéder de manière globale (échelle planétaire) grâce au protocole phare du Web : *HTTP* [82]. Ce protocole est spécifiquement conçu pour accéder à des représentations d'une ressource, notamment grâce à ses mécanismes de *négociation de contenu*. Par exemple, une même

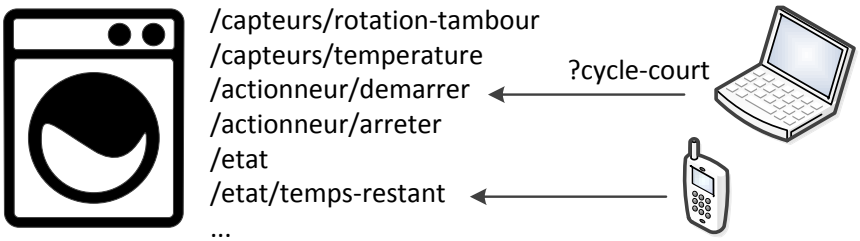


Figure 2.11 – Exemples de services Web pour un lave-linge.

ressource peut être accédée dans des langues différentes ou dans des formats différents, en fonction de ce que l'utilisateur désire. Cette philosophie basée sur une ressource et ses multiples représentations est dénommée *representational state transfer*, ou *REST*.

Aussi, une fois les objets capables de communiquer au travers du réseau Internet, une extension naturelle consiste à utiliser le Web pour abstraire les objets hétérogènes sous la forme de ressources directement manipulables par les utilisateurs (accès aux capteurs, contrôle des actionneurs, etc.). De cette façon, l'Internet des objets devient le Web des objets [83], un grand réseau de ressources que les utilisateurs peuvent utiliser pour interagir avec le monde physique littéralement de la même façon qu'avec le monde virtuel. En outre, du fait de sa popularité, le Web s'accompagne de nombreuses infrastructures utiles du point de vue de l'Internet des objets : recherche de ressources (moteurs de recherche, annuaires), réplication de ressources (réseaux de distribution de contenu, caches), conservation de l'historique des ressources ou encore la notification des changements (flux de syndication, Web *hooks*), etc.

L'accès à l'identité de l'objet, la collecte des informations produites et la transmission des actions à effectuer peuvent se représenter grâce à des services Web. Les utilisateurs ou d'autres objets peuvent alors invoquer ces services pour obtenir des informations ou pour commander des objets, comme le montre la Figure 2.11. L'utilisation de services Web a pour avantage de rendre possible la création de *mashups* basées sur les données fournies par les objets. Dans la terminologie propre au monde du Web, une *mashup* est une application Web qui exploite les capacités offertes par différents services ou sources de données, par exemple une application combinant un service de cartographie avec des statistiques sur les tremblements de terre [7]. Ainsi, le Web des objets rend possible la création de *physical mashups* [84] qui mêlent aussi bien des données issues des objets que des données existantes, déjà accessibles sur le Web : réseaux sociaux, cartes géographiques, encyclopédies, informations météorologiques ou encore les informations diffusées par les gouvernements dans le cadre des politiques *open data* [8]. En pratique, on trouve aujourd'hui deux technologies majeures pour la création de services Web [83] :

7. *http://www.oe-files.de/gmaps/eqmashup.html* (accédé le 01/08/2014).

8. Les politiques *open data* sont des initiatives gouvernementales visant à diffuser les données produites par les différentes infrastructures gouvernementales (transports publics, trafic routier, analyse de la pollution, statistiques diverses, etc.).

Les services WS-* Ces services possèdent une interface (entrées, sorties, propriétés) décrite dans un format manipulable par des machines et sont invoqués au moyen de messages *SOAP* transportés sur *HTTP*. Cette technologie de services est intéressante de par les mécanismes qu'elle propose (description des interfaces, par exemple) et ses extensions standardisées (chiffrement, etc.). En outre, les services *WS-** s'intègrent dans une architecture orientée service plus vaste, avec ses propres technologies d'annuaire (*UDDI*) et d'orchestration de service (*WS-BPEL*, *WS-Coordination*, *WS-Transaction*). Toutefois, il s'agit d'une technologie difficile à implémenter sur des appareils possédant peu de ressources, du fait de sa lourdeur [85].

Les services RESTful Ces services exploitent directement le principe de représentation des ressources qui sous-tend la conception du protocole *HTTP* (*REST*). Ici, ce sont les messages *HTTP* qui sont utilisés directement, au lieu d'un protocole supplémentaire, *HTTP* permettant notamment de décrire des actions telles que la récupération (message *GET*), la création (message *POST*), la modification (message *PUT*) ou la suppression (message *DELETE*) d'une ressource [86]. En comparaison des services *WS-**, les services *RESTful* sont plus flexibles, permettant l'utilisation de nombreux formats pour transférer les paramètres lors de l'invocation du service et pour récupérer les résultats. En conséquence, la possibilité d'utiliser des formats légers simplifie leur implémentation pour des systèmes embarqués limités en ressources [85]. Cependant, les services *RESTful* ne possèdent pas de mécanisme standard pour décrire l'interface du service ou pour composer les services entre eux.

Dans tous les cas, *HTTP* est nécessaire, ce qui requiert d'intégrer un serveur Web directement au sein des objets. Toutefois, les serveurs *HTTP* modernes ne sont pas utilisables directement, du fait de leur complexité. En effet, ces serveurs sont généralement conçus pour servir des milliers de connexions simultanées et implémentent la totalité du protocole *HTTP*, ce qui couvre un grand nombre de cas d'utilisations (mise en cache, contrôle d'accès, négociation de contenu, etc.) [85]. Aussi, les premières approches pour le support des services Web dans les réseaux de capteurs se sont basées sur des proxys pour représenter les différents nœuds sous la forme de ressources. Concrètement ce rôle de proxy est assuré par une base qui intègre le serveur Web ainsi que la logique d'exécution des services. Lorsque l'un des services est invoqué, le proxy récupère les données depuis les capteurs ou transmet les ordres aux actionneurs appropriés, au moyen du protocole de communication spécifique au réseau de capteurs ou d'un protocole allégé d'invocation de service [87]. Cette méthode a été utilisée aussi bien pour des services *RESTful* [87] que pour des services *WS-** [88]. Cependant, toutes les invocations de services passent par le proxy, qui peut se retrouver rapidement engorgé si le nombre d'utilisateurs est grand. Aujourd'hui, ces approches tendent à être remplacées par des proxys hébergés dans le *cloud* [89, 90] ou au sein de serveurs spécialisés.

Par la suite, il a été proposé d'intégrer directement des serveurs Web au sein des nœuds [91] pour permettre, d'une part, aux différents objets de gagner en autonomie en ne dépendant plus des proxys et, d'autre part, de répartir la charge à travers l'ensemble du réseau en faisant disparaître les goulots d'étranglement du réseau. Cependant, comme les serveurs Web actuels ne peuvent pas être utilisés en l'état, ils sont généralement adaptés aux spécificités des systèmes restreints en ressources [92] et prennent le nom de *serveurs Web embarqués*. Adapter le protocole *HTTP* à des appareils fortement limités en ressources présente toutefois de nombreuses difficultés, d'où de nombreuses limitations en pratique pour de tels systèmes ; par exemple, la possibilité de ne servir que des pages statiques [92]. Pour pallier à ces problèmes, le protocole *CoAP* a été conçu comme un remplaçant d'*HTTP* pour l'implémentation des services *RESTful* dans les systèmes restreints. *CoAP* est significativement plus léger que *HTTP*, notamment grâce à un format de message binaire et concis, qui réduit la taille des messages d'environ 85% en moyenne [93]. En outre, là où *HTTP* implique l'utilisation de *TCP* en pratique, *CoAP* exploite *UDP* comme protocole de transport. Pour pallier à l'absence de garantie d'*UDP* quant à la fiabilité (pas de contrôle de congestion ou de respect de l'ordre des paquets), *CoAP* implémente des mécanismes optionnels permettant de détecter la perte de messages et de les retransmettre si nécessaire.

2.1.3.2 Le Web sémantique des objets

Originellement, le Web a été conçu pour diffuser et présenter du contenu à destination des êtres humains : agréable à lire (texte formaté, documents multimédias), facile d'accès et interactif. Cependant, cette représentation a un impact sur la capacité des machines à raisonner à partir de ces informations et rend de fait difficile la réalisation de certains scénarios basés sur l'extraction automatique d'informations depuis des pages Web. Le concept de *Web sémantique* a alors été introduit comme une extension du Web actuel, dans laquelle les informations sont spécifiquement structurées, annotées et contextualisées de façon à ce que les machines puissent en manipuler plus facilement le sens et, de fait, améliorer les interactions avec les êtres humains [94]. De ce point de vue, le Web sémantique est une continuité naturelle pour le Web des objets, aboutissant à la création d'un *Web sémantique des objets*. Celui-ci permet notamment de faire le lien entre la vision orientée Internet et la vision orientée sémantique dont nous avons parlé en début de ce chapitre.

Spécifiquement, le Web sémantique se compose d'un ensemble de technologies dédiées à des aspects divers : description des connaissances et des relations qui les unissent, partage et manipulation de ces connaissances, raisonnement automatisé, etc. Ces technologies sont standardisées par le *W3C*[9] et s'organisent en couches, comme présenté sur la Figure 2.12 ; auxquelles le terme *semantic web stack* fait souvent référence. Parmi

9. Le *W3C* (*World Wide Web Consortium*) est un organisme de standardisation pour les couches applicatives du réseau Internet, notamment le Web.

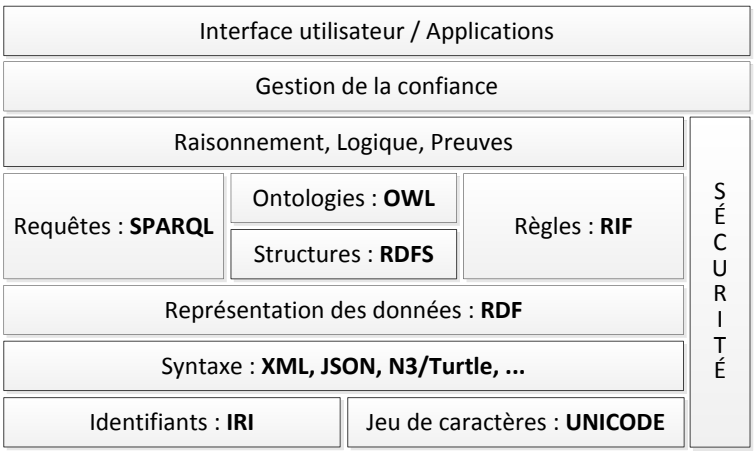


Figure 2.12 – La *semantic web stack*.

les couches basses, on trouve notamment les technologies fondamentales permettant d'identifier les ressources (*URI* et *IRI*) et d'encoder des caractères (*Unicode*). Ces technologies servent de base à l'expression du modèle de données de référence du Web sémantique, le *Resource Description Framework* (*RDF*). Celui-ci est un modèle abstrait qui permet de représenter des connaissances structurées sous la forme d'ensembles de connaissances simples appelées *triplets RDF*. Ces ensembles de connaissances, ou *documents RDF*, peuvent ensuite être sérialisés dans des formats d'échange concrets, ou *syntaxes*, tels que *XML* ou *JSON*. Concrètement, un triplet *RDF* est composé des termes suivants :

— un *sujet*, qui indique à quelle ressource se rapporte cette connaissance ;

— un *prédicat*, qui indique quelle propriété du sujet est concernée ;

— un *objet*, qui représente la valeur de la propriété.

Ces paramètres peuvent alors prendre trois types de valeurs :

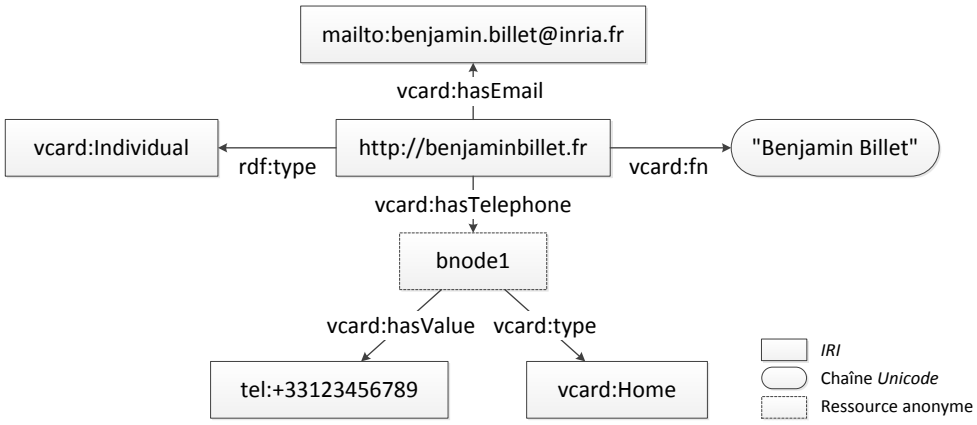
— une *IRI*, qui peut représenter aussi bien un sujet, un prédicat (nom *URN* d'une propriété) ou un objet (liens vers d'autres ressources) ;

— une chaîne *Unicode* (utilisable uniquement comme objet), qui peut être annotée d'un type littéral ou d'une langue ;

— une ressource anonyme (*blank node* ou *bnode*) qui peut être utilisée pour représenter une connaissance dont le nom est inconnu ou non spécifié. Ces ressources décrivent des connaissances locales au document *RDF*, et peuvent être utilisées aussi bien pour qualifier un sujet qu'un objet.

Les différents triplets *RDF* sont reliés entre eux, soit au moyen des ressources anonymes (liens internes au document *RDF*), soit au moyen des *IRI* (liens entre différents documents *RDF*). Implicitement, les ensembles de triplets *RDF* forment un graphe décrivant les relations entre les connaissances, comme illustré sur la Figure 2.13.

RDF sert de modèle générique pour la représentation des connaissances dans le Web sémantique. Au-delà de cette représentation uniforme des données, l'objectif du Web




```
(http://benjaminbillet.fr rdf:type vcard:Individual)
(http://benjaminbillet.fr vcard:fn "Benjamin Billet")
(http://benjaminbillet.fr vcard:hasEmail mailto:benjamin.billet@inria.fr)
(http://benjaminbillet.fr vcard:hasTelephone _:bnode1)
(_:bnode1 vcard:hasValue tel:+33123456789)
(_:bnode1 vcard:type vcard:Home)
```

Figure 2.13 – Extrait de document *RDF* pour une carte de visite.

sémantique consiste à fournir les technologies nécessaires pour (i) structurer et manipuler les triplets *RDF*, ceux-ci étant disséminés sur le Web, (ii) raisonner automatiquement à partir de ces connaissances tout en considérant les aspects liés à la confiance et à la sécurité, et (iii) fournir aux utilisateurs les outils applicatifs nécessaires pour tout cela, à l'image des navigateurs Web qui permettent aujourd'hui d'accéder facilement aux représentations des ressources Web. Plusieurs de ces mécanismes ne sont encore ni clarifiés ni spécifiés, le *W3C* ayant travaillé principalement sur la structure et la manipulation des connaissances, au travers des quatre standards suivants :

— Le langage *SPARQL*, qui permet d'exprimer des requêtes sur les graphes *RDF*, et dont la syntaxe s'inspire en partie du langage de requête *SQL*. *SPARQL* fournit en outre des constructions adaptées pour la recherche de triplets *RDF* correspondant à des motifs donnés. En outre, *SPARQL* introduit un protocole censé permettre, à terme, de rechercher, manipuler et transformer des données *RDF* disponibles sur le Web.

— Les dialectes *RIF*, qui ont été pensés à l'origine pour exprimer des règles logiques (*RIF-BLD*) et des règles de production (*RIF-PRD*) dans le cadre du modèle *RDF*. Aujourd'hui, le travail du *W3C* sur *RIF* consiste à introduire des dialectes permettant à des systèmes existants de faire la traduction entre leurs langages propres et les langages *RIF*, de façon à ce que ces systèmes puissent échanger des règles entre eux sous une forme unifiée [95]. À l'heure actuelle, seules les règles logiques bénéficient d'un tel dialecte (*RIF-FLD*).

— Le langage *RDFS* (*RDF Schema*), qui permet de décrire la structure des connaissances et les relations qui les unissent (types, taxonomies, etc.) ; par exemple, spécifier qu'un sujet est d'un certain type.

— Le langage ontologique [10] *OWL* qui introduit le vocabulaire pour décrire les connaissances de manière plus sophistiquée que *RDFS* : relation d'équivalence, composition (par exemple, « une voiture possède quatre roues »), exclusion (par exemple, « un chat n'est pas un chien »), etc.

Les ontologies sont particulièrement utiles dans le cadre de la vision orientée sémantique de l'Internet des objets. En effet, elles permettent de décrire les objets, leurs caractéristiques, leurs capteurs et leurs actionneurs au moyen de vocabulaires unifiés, ce qui permet de réduire l'impact de l'hétérogénéité [96]. De la même façon, les ontologies peuvent aider à mieux traiter l'échelle de l'Internet des objets, notamment en modélisant les méthodes d'approximation ou d'estimation propres à des capteurs, à des phénomènes physiques ou à des domaines particuliers [96]. De cette façon, il devient possible de réduire la taille des flux de mesures (approximation) ou de déterminer des équivalences moins coûteuses ; par exemple, en sélectionnant des sous-ensembles représentatifs d'objets pour un objectif donné.

Les ontologies pour l'Internet des objets dérivent en partie des travaux menés dans le cadre des réseaux de capteurs et d'actionneurs sans fil, notamment ceux de l'*OGC* (*open geospatial consortium*) [11] sur la standardisation des interfaces de services Web pour les capteurs (*sensor web enablement*, ou *SWE*). *SWE* a pour but de permettre l'utilisation uniforme des capteurs, ce au travers de mécanismes interopérables de découverte, d'accès et de déploiement [97]. *SWE* introduit notamment *SensorML*, pour la description des entrées et les sorties des capteurs et des actionneurs, et *O&M* (*Observations and Measurements*) pour la description des mesures et des observations produites par les capteurs (type, domaine valeur, etc.) [97]. Pour faire le lien avec *RDF*, le *W3C* travaille sur l'ontologie *SSN* (*semantic sensor network*), basée sur les concepts de *SWE* et compatible avec le Web sémantique (*OWL*). Cette ontologie permet notamment de modéliser les capteurs sous plusieurs perspectives (phénomènes mesurés, caractéristiques techniques, entrées-sorties, etc.) [98].

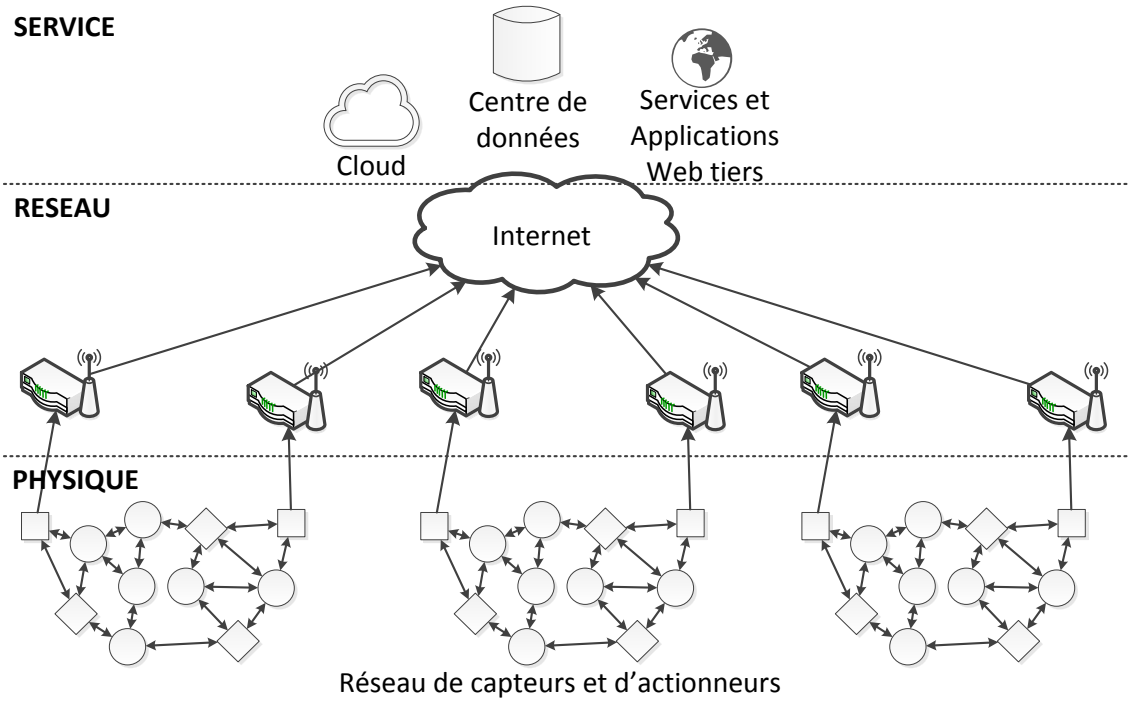


Figure 2.14 – Une architecture hiérarchique pour l'Internet des objets.

2.1.4 Gestion de l'échelle de l'Internet des objets

Comme nous l'avons déjà mentionné, l'échelle de l'Internet des objets est une problématique majeure, l'extension des réseaux de capteurs à l'échelle mondiale impliquant de très grands volumes de données et nécessitant alors une réflexion particulière quant à la façon dont ces données sont obtenues, stockées et traitées. *IrisNet* [99, 100] est l'une des premières approches à avoir questionné la mise en œuvre des réseaux de capteurs à échelle planétaire. Pour ce faire, le réseau de capteurs est découpé hiérarchiquement en deux types de composants : (i) des *sensing agents*, qui sont des interfaces génériques pour l'accès aux données des capteurs d'une zone ou d'un groupe spécifique, et (ii) des *organizing agents* qui sont des bases de données distribuées stockant les informations collectées depuis les *sensing agents* par les différentes applications. Cette hiérarchisation est rendue possible, car, tout comme dans les réseaux *RFID*, il existe plusieurs niveaux dans les réseaux de capteurs à grande échelle [90, 101, 102], comme illustré sur la Figure 2.14 :

— un niveau composé d'objets physiques capables d'interagir entre eux directement, comme le font les lecteurs et les puces *RFID* ;

10. En informatique, une ontologie est un ensemble structuré de termes, de concepts et de relations destiné à représenter la sémantique d'un champ de connaissance. Les ontologies sont typiquement utilisées pour permettre aux machines de raisonner automatiquement à propos du champ de connaissance en question.

11. L'*Open Geospatial Consortium* est un consortium international qui développe des standards ouverts et des spécifications pour accroître l'interopérabilité entre les domaines et les systèmes qui exploitent des données géographiques (géomatique).

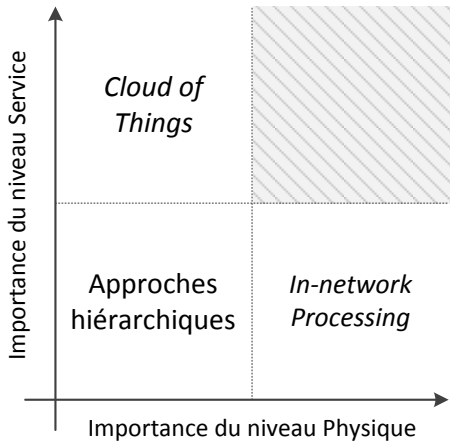


Figure 2.15 – Le positionnement des approches de réseaux de capteurs à grande échelle.

— un niveau formé de l'infrastructure du réseau Internet et des passerelles qui font le lien avec les protocoles des réseaux de capteurs ;

— un niveau constitué de services globaux, le plus souvent fournis par un tiers, et grâce auxquels les utilisateurs interagissent avec l'Internet des objets (équivalent des *EPC Network Services*) : réseaux sociaux, visualisation et fouille de données, stockage et synchronisation, partage et découverte de données, etc.

Cette structure hiérarchique est exploitée par de nombreuses approches pour supporter l'échelle de l'Internet des objets, les différences portant sur l'importance donnée au niveau physique et au niveau service comme montré sur la Figure 2.15. On trouve, tout d'abord, des approches telles que le *cloud of Things* [103], qui limitent les capacités d'actions des objets physiques au minimum en déportant la logique applicative vers des services tiers hébergés dans le *cloud* ou dans des centres de données. Les capteurs collectent des informations et les transfèrent vers ces services qui les stockent et les traitent tout en fournissant des interfaces sophistiquées aux utilisateurs pour leur permettre de visualiser et d'explorer leurs données ainsi que celles des autres. Il s'agit, globalement, d'abstraire et de virtualiser les capteurs et les actionneurs sous la forme de services (*things as a service*, ou *TaaS*), ce pour atténuer les problèmes d'hétérogénéité et profiter, en outre, de l'élasticité et de la dynamicité offertes par le *cloud* [90, 104, 105]. Ces approches rejoignent l'idée d'un *sensor cloud* [106], où les capteurs physiques sont abstraits sous la forme de capteurs virtuels pouvant dès lors être regroupés et manipulés de manière transparente par les utilisateurs. Toutefois, si le *cloud* est élastique, aussi bien en matière de stockage que de calculs, les coûts énergétiques liés à la communication entre les nœuds et les serveurs sont à considérer, étant donné que les liaisons sans fil sont très consommatrices d'énergie [67]. De plus, cette approche n'est pas adaptée aux nœuds mobiles, ces derniers étant sujets à une connectivité intermittente. Enfin, la centralisation des données collectées, qui plus est par des services tiers, pose des problèmes en matière de respect de la vie privée.

À l'opposé se situent les approches dites de *in-network processing*, qui mettent l'accent sur les interactions autonomes entre objets physiques [107]. Il s'agit ici de laisser les nœuds interagir directement, de la même façon que les lecteurs et les étiquettes *RFID*, sachant que :

— même les nœuds les plus faibles possèdent des capacités matérielles suffisantes pour effectuer des tâches simples ;

— tous les objets physiques n'ont pas forcément vocation à communiquer entre eux, les scénarios étant généralement restreints géographiquement : la maison, le bureau, l'entreprise, la ville, etc. Au sein de ces espaces, les objets peuvent interagir entre eux sans avoir besoin de communiquer au niveau global.

Les nœuds étant autonomes et responsables de leurs propres données, ils peuvent dès lors effectuer des traitements divers avant de les transmettre à d'autres nœuds ou à des services tiers. En effet, dans ces approches, les services tiers ne sont pas exclus, mais considérés comme des *super-nœuds* capables d'effectuer des traitements plus complexes (décodage vidéo, reconnaissance vocale, fouille de données, etc.). Historiquement, les approches *in-network processing* ont été utilisées pour réduire le volume des données échangées, par exemple en filtrant, compressant ou agrégeant les informations localement. Cela a été notamment étudié dans le contexte de l'analyse structurelle des bâtiments [108], où des capteurs de vibration produisent de très grandes quantités de mesures, dans les réseaux *RFID*, pour le nettoyage des évènements produits par les lecteurs [109], ou encore dans les systèmes de détection d'intrusion, où le nombre d'évènements est élevé [110]. Les interactions directes entre les nœuds présentent l'avantage de réduire significativement la consommation énergétique, étant donné que seules les informations nécessaires sont transmises [111]. De plus, la logique applicative et le contrôle des actionneurs sont hébergés au sein du réseau et n'impliquent des services tiers que lorsque cela est nécessaire (partage des données ou traitement lourd, par exemple). Par ailleurs, comme les appareils n'ont pas besoin de transmettre l'ensemble de leurs informations à des services tiers, les problématiques de vie privée sont atténuées. Cependant, la mise en place et la gestion de tels réseaux est plus complexe, du fait de la forte hétérogénéité.

Entre ces deux extrêmes, différentes approches hiérarchiques ont été proposées pour atténuer les défauts propres aux approches *cloud of Things* et *in-network processing*. Notamment, il a été suggéré d'utiliser un niveau intermédiaire composé de machines puissantes et continuellement alimentées en énergie (*surrogates*), disséminées dans l'environnement [112]. Ces dernières peuvent être utilisées par les objets les plus proches pour déléguer d'éventuels traitements complexes. La notion de *cloudlets* [113] est similaire, une *cloudlet* étant une machine destinée à faire le lien entre les appareils du niveau physique et le *cloud*, notamment en prenant le relai pour le traitement des tâches lorsque la connexion au *cloud* est perdue [114]. Ces approches tentent globalement de rapprocher physiquement le niveau service et les appareils finaux, de façon à assurer la continuité des différents services offerts. Ce concept est parfois désigné sous le terme *fog computing* [115].

En pratique, ces approches hiérarchiques tiennent compte des interactions locales et globales et permettent donc de représenter des architectures complexes composées de plusieurs niveaux d'interactions et de calcul ; par exemple dans le cadre de la gestion des objets sous-marins [116] ou des nano-objets [117].

2.2 Systèmes de traitement de flux : une analyse de l'existant

Comme nous l'avons mentionné dans l'introduction de ce mémoire, l'Internet des objets renforce la présence de données évoluant régulièrement au cours du temps et devant être traitées en continu, du fait de la présence massive de capteurs. À cette fin, il est raisonnable d'envisager la représentation des données sous la forme de flux et l'utilisation des modèles de traitement qui y sont associés. D'autant plus que, d'après les estimations, le trafic généré par l'Internet des objets sera colossal et nécessitera l'exécution d'opérations complexes sur des flux de grande taille et de débit élevé, ce pour des durées très longues [29].

En quelques mots, un flux est une suite d'éléments discrets partageant des caractéristiques communes, de telle sorte que l'on choisit de les considérer virtuellement comme étant un même ensemble de données ; par exemple la température à un endroit donné. Dans la plupart des cas, la fin d'un flux n'est pas prédictible à priori, tout comme le nombre d'éléments contenus dans le flux. Chacun de ces éléments est produit à un instant donné et, dans de nombreux cas, l'intérêt que l'on porte à cet élément décroît au cours du temps (c.-à-d., les données anciennes sont potentiellement moins utiles que les données récentes).

De manière générale, le travail de représentation et de traitement des flux est effectué au travers d'un *système de gestion de flux de données* (*data stream management system*, ou *DSMS*), aussi appelé *système de traitement de flux* (*stream processing system*, ou *SPS*) [30], qui se présente sous la forme d'une infrastructure permettant aux utilisateurs d'exprimer des traitements continus sur les flux [118]. Concrètement, le *DSMS* possède un rôle similaire au *système de gestion de base de données* (*data base management system*, ou *DBMS*), la différence principale étant conceptuelle : en lieu et place de manipuler des ensembles finis de données, le *DSMS* manipule des flux théoriquement infinis.

En pratique, cette différence a un impact important sur la manière dont les données sont traitées, comme le résume la Table 2.2 [39]. Tout d'abord, il s'agit d'un changement radical dans la façon de traiter les données. Dans un *DBMS*, les ensembles de données sont finis et persistants tandis que les traitements qui y sont appliqués sont volatiles, en cela qu'ils n'ont plus de sens lorsque l'ensemble est traité. Dans un *DSMS*, au contraire, les flux étant des ensembles dont la taille est théoriquement infinie, les données perdent leur caractère persistant (elles n'ont de sens qu'à l'instant où elles sont produites) tandis que les requêtes deviennent persistantes du fait de leur exécution permanente [39].

	DBMS	***DSMS***
Données	Relations persistentes	Flux, fenêtres temporelles
Accès aux données	Aléatoire	Séquentiel, passe unique
Type de mise à jour	Arbitraire	Ajout en fin seulement
Fréquence de mise à jour	Relativement basse	Haute, par rafales
Traitement	Dirigé par les requêtes* (*pull-based*)	Dirigé par les données (*push-based*)
Requêtes	Executée une seule fois	Continue
Plan de requête*	Fixe	Adaptatif
Optimisation	Une seule requête*	Multi-requête*
Résultat	Exact	Exact ou approché
Latence	Relativement haute	Faible

* À noter que le terme « requête » pourrait être remplacé par « tâche », dans un contexte autre que celui des bases de données.

Table 2.2 – Différences entre un *DBMS* et un *DSMS*.

Cette particularité modifie la façon dont les données sont consommées, un *DBMS* allant directement chercher les données dans des ensembles structurés (*pull-based*) et un *DSMS* ne pouvant que réagir à l'arrivée de nouvelles données (*push-based*), au fur et à mesure de leur production. Qui plus est, comme les données sont produites en grande quantité, à un rythme soutenu ou en rafales (c.-à-d. des pics de production de données) et pendant de longues périodes de temps, les *DSMS* utilisent des techniques propres aux fouilles de données (*data mining*) et peuvent fournir, si les contraintes techniques le nécessitent, des réponses approchées.

Au-delà des données évoluant au cours du temps, pour lesquelles la structuration en flux est parfaitement naturelle, le traitement continu présente un intérêt non négligeable pour le traitement de données en général. En effet, il s'agit d'un paradigme de calcul pouvant gérer de grands volumes de données et s'adapter aux variations de charge et de disponibilités des ressources [30]. En outre, les systèmes de traitement de flux sont le plus souvent flexibles et peuvent être étendus pour toutes sortes de calculs, aussi bien sur des données structurées (par exemple, lorsque celles-ci s'accompagnent d'un schéma de données) que sur des données non structurées (données audio-vidéo, image et texte brut).

Globalement, comme le montre la Figure 2.16, un système de gestion de flux de données met en œuvre le concept de *producteur de données* (ou *source de données*), c'est-à-dire une entité qui produit des informations au cours du temps sous forme d'éléments. Ces informations partagent des caractéristiques communes et on choisit alors de les considérer non plus comme une suite discrète de données, mais comme un *flux*. Ces informations sont alors récupérées par des *consommateurs de données* qui vont effectuer

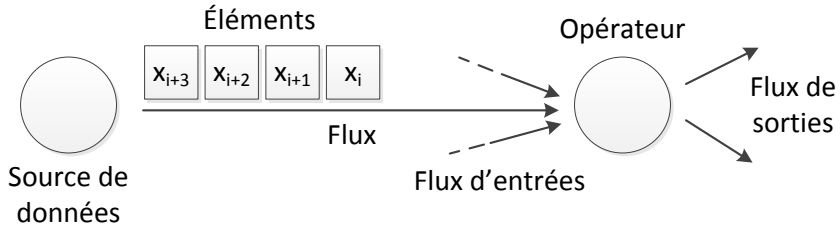


Figure 2.16 – Le concept de flux.

toutes sortes d'opérations à partir de celles-ci (transformation, calcul, stockage, affichage, etc.) et, éventuellement, produire de nouveaux flux.

Historiquement, la question du traitement continu des flux émerge de plusieurs domaines tels que la fouille de données, le calcul parallèle et distribué, les bases de données relationnelles, l'informatique décisionnelle (*business intelligence*) et les entrepôts de données. De là, il est assez difficile d'établir des catégories pour classer les systèmes de gestion de flux, en cela qu'ils se différencient en raison de leurs origines, leurs philosophies, leurs architectures et leurs langages d'expression de tâches [30, 41]. Dans la suite de cette section, nous distinguons les *DSMS* qui exploitent un modèle de données dérivé du modèle relationnel (*DSMS* relationnels) de ceux qui n'exploitent pas de modèle de données particulier (*DSMS* génériques).

2.2.1 Systèmes de traitement de flux issus du modèle relationnel

L'idée du traitement continu sous forme de requêtes prend racine dans le domaine des systèmes de gestion de bases de données, et plus spécifiquement avec les trois types de bases de données suivants [119, 120] :

— Les *bases de données actives* et les *triggers*, qui permettent aux utilisateurs de définir des actions à exécuter lorsque certains changements sont appliqués à une table (ajout, suppression, etc.).

— Les *bases de données temporelles*, ou *base de données historiques*, qui enregistrent chaque changement effectué dans les tables et permettent à l'utilisateur d'effectuer des requêtes sur ces états passés.

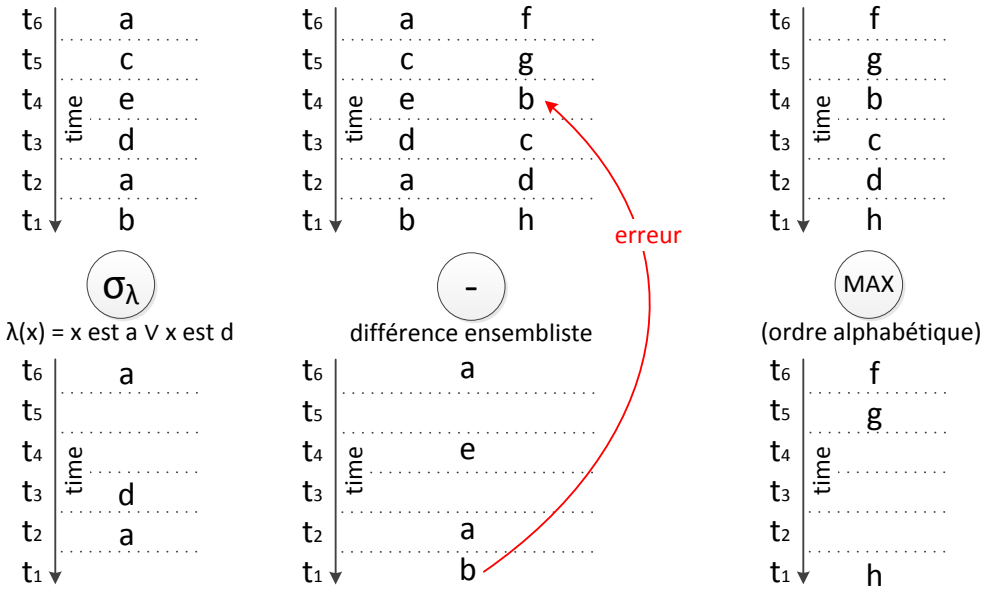
— Les *bases de données séquentielles*, qui introduisent la notion de séquence, c'est-à-dire un ensemble de n-uplets auquel est associé une relation d'ordre [120], et formalisent les opérateurs propres à ces structures.

De là, il a été proposé d'étendre le modèle relationnel défini par Edgar Frank Codd [121] pour y introduire la notion de flux, c'est-à-dire des séquences ordonnées dont la taille est infinie (où, en pratique, imprédictible), composées d'éléments produits en continu par la source du flux. De manière analogue aux relations, les flux sont structurés et définissent un schéma auquel chaque élément contenu dans le flux se conforme [39]. En

Un *n-uplet* est une collection ordonnée de n termes, notée $(a_1, \cdots, a_n)$ pour tout $n > 0$. Pour rappel, *singleton* désigne un *1-uplet*, *couple* (ou *paire*) un *2-uplet*, *triplet* un *3-uplet*, etc.

Une *relation* est un ensemble de n-uplets respectant une structure commune (appelée *schéma*). Formellement, étant donné n ensembles $S_1, S_2, \cdots, S_n$, R est une relation sur ces n ensembles si R est un ensemble de n-uplets (*tuples*) où le premier élément appartient à S_1, le second à S_2 et ainsi de suite. Ce qui revient à dire que R est un sous-ensemble du produit cartésien $S_1 \times S_2 \times \cdots \times S_n$ [121].
Les propriétés (nom, type, contraintes, etc.) de chaque ensemble S_i, ou *i-ème attribut* de R, composent alors le *schéma* de R auquel chaque n-uplet de R se conforme [122].

Encart 2.1 – Rappel sur les définitions de n-uplet et de relation dans l'algèbre relationnelle.



(a) Projection (b) Différence ensembliste (c) Maximum

Figure 2.17 – Quelques exemples d'opérations continues.

outre, chaque élément est associé à une information temporelle explicite (timestamp) ou implicite (temps d'arrivée) [123], qui définit la relation d'ordre de la séquence.

2.2.1.1 Traitement continu

Un traitement continu quelconque s'effectue sous la forme d'un opérateur qui implémente une fonction spécifique et qui consomme un ou plusieurs flux en entrée pour produire un ou plusieurs flux en sortie. Parmi les opérateurs relationnels, certains peuvent s'appliquer naturellement aux flux de données. Par exemple, la *sélection*, qui construit une nouvelle relation R' à partir des n-uplets d'une relation R qui satisfont une expression logique λ, peut être exécutée au fur et à mesure de la réception des n-uplets. Il suffit,

Définition 2.1 Opérateur monotone (algèbre relationnelle) Soit deux relations R_1 et R_2. Un opérateur O est dit monotone si et seulement si, lorsque $R_1 \subseteq R_2$, on a : $O(R_1) \subseteq O(R_2)$, ou $O(R_1 \cup R_2) \supseteq O(R_1) \cup O(R_2)$ [125].

Définition 2.2 Opérateur continu monotone La notion d'opérateur monotone s'applique aux opérateurs continus, relativement à la relation d'ordre. Soit s_1 un flux et s_1^m la séquence des m premiers éléments ordonnés $(x_{11}, \cdots, x_{1m})$ de s_1. On dira qu'une séquence $s_2^n = (x_{21}, \cdots, x_{2n})$ est une *préséquence* de s_1^m, notée $s_2^n \sqsubseteq s_1^m$, si et seulement si (1) $n \leq m$ et (2) $\forall i \in [\![1, n]\!]$ on a $x_{1i} = x_{2i}$ [124].
Dans ce contexte, la monotonie avec respect de l'ordre se traduit par : $s_2^n \sqsubseteq s_1^m \implies O(s_2^n) \sqsubseteq O(s_1^m)$.

Définition 2.3 Opérateur non bloquant Un opérateur O est dit non bloquant si et seulement si $\forall n$, on a $O(s^n) = O(s)^n$ [124], avec $O(s)$ le flux résultant de l'application de l'opérateur O sur le flux s.

Implication du caractère non bloquant des opérateurs monotones Soit un opérateur monotone O, un flux s et deux séquences s^m et s^n telles que $s^n \sqsubseteq s^m$. On a $O(s^n) = O(s^n)^n = O(s^m)^n$, ce qui, si l'on considère qu'un flux s est une séquence infinie s^∞, équivaut à dire que $O(s^n) = O(s)^n$. Conformément à la Définition 2.3, on en conclut que *tout opérateur monotone peut être implémenté sous forme non bloquante* [124].

Encart 2.2 – Démonstration du caractère non bloquant des opérations monotones.

pour cela, d'appliquer λ à tout nouvel élément reçu i et de n'écrire i sur la sortie que si la proposition $\lambda(i)$ est vraie. Toutefois, certaines opérations ne peuvent pas être appliquée de manière continue et nécessite d'avoir récupéré l'ensemble complet des n-uplets pour pouvoir être exécutée. C'est par exemple le cas de la *différence ensembliste* qui, à deux ensembles donnés A et B, produit l'ensemble $\{x \in A | x \notin B\}$. L'exemple présenté sur la Figure 2.17 montre comment la différence ensembliste continue introduit une inconsistance à partir de $t = t_4$.

On parle donc généralement d'*opérateurs bloquants* et d'*opérateurs non bloquants*, selon qu'il soit nécessaire ou non de récupérer l'intégralité du flux avant d'effectuer l'opération. Intuitivement, l'exemple de la différence ensembliste permet d'identifier que les opérations non bloquantes sont celles qui, lorsqu'un nouveau n-uplet est traité à l'instant t, n'ont pas besoin de réviser les n-uplets résultats qu'elles ont produit à l'instant $t - 1$. Formellement, il a été démontré que tout opérateur monotone (voir Définition 2.1) peut être implémenté sous une forme non bloquante [124], ce qui comprend notamment la *projection*, la *sélection*, la *jointure*, l'*union* et l'*intersection* d'ensembles et le *produit cartésien*.

Comme on peut l'imaginer, l'implémentation des opérateurs a un impact important sur leur nature non bloquante. Par exemple, on retrouve des implémentations continues d'opérateurs d'agrégation non monotones dont la cardinalité de l'ensemble résultat vaut

1, tels que la *moyenne* (*AVG*) ou la *somme* (*SUM*). La Figure 2.17 présente un exemple avec l'opérateur *MAX*, où chaque nouveau n-uplet en entrée provoque l'écriture du nouveau maximum en sortie, si ce dernier a changé. Dans ce cas particulier, le maximum est traité comme une information qui évolue au cours du temps (p. ex. comme la température), et dont chaque résultat intermédiaire est produit en sortie de l'opérateur ; le maximum le plus récent étant celui qui présente le plus d'intérêt [126].

Cette approche se généralise à tous les opérateurs bloquants, au travers de la notion de fenêtre. Globalement, une fenêtre construit un ensemble fini à partir d'un flux, pour permettre à un opérateur bloquant de travailler sur le flux morceau par morceau. Si l'on revient au comportement de l'opérateur *MAX* de la Figure 2.17, on peut considérer que ce dernier possède une fenêtre qui construit, à partir du flux, des ensembles contenant un seul élément. Si la notion de fenêtre existait déjà auparavant [127], c'est au travers du prototype de *DSMS STREAM* [128, 129] et de son langage de requête *CQL* (*continuous query language*) [118] que celle-ci a été formalisée en tant qu'opération de transformation des flux vers des relations.

2.2.1.2 Flux vers relation

Globalement, les fenêtres permettent de tronçonner le flux pour ne pas avoir à le stocker entièrement avant de le traiter et sont donc primordiales pour l'implémentation des opérateurs bloquants. Par ailleurs, cette approche se justifie aussi dans le cadre des opérateurs non bloquants, étant donné que l'utilisateur est souvent intéressé par les données les plus récentes [39].

De manière générale, les opérateurs de fenêtres se classent en deux grands types, qui définissent la façon dont celles-ci sélectionnent les éléments du flux :

— les *fenêtres logiques* (ou *time-based*), qui sont définies par un intervalle de temps (p. ex. les cinq dernières minutes) ;

— les *fenêtres physiques* (ou *count-based*, *tuple-based*), qui sont définies par un intervalle d'éléments (p. ex. les 10 derniers éléments).

Les fenêtres sont caractérisées par deux points qui définissent la direction de la fenêtre : un *point de départ* et un *point d'arrivée*. Il peut s'agir de points dans le temps, pour les fenêtres logiques, ou de rang (i-ème élément) pour les fenêtres physiques. Ces points peuvent être fixes ou mobiles, ce qui nous donne quatre combinaisons possibles [12]. Lorsque les points de départ et d'arrivée sont fixes, on parle simplement de *fenêtre fixe* (*fixed window*) en cela qu'elles extraient une portion fixe du flux. Les *fenêtres glissantes* possèdent deux points mobiles, spécifiés relativement à l'instant présent. Contrairement à leurs homologues fixes, le contenu d'une fenêtre glissante change au cours du temps, au

12. Théoriquement, les points mobiles peuvent avancer ou reculer dans le temps [39], mais nous choisissons d'ignorer le cas du recul, étant donné qu'il n'est pas possible de revenir sur des éléments déjà traités.

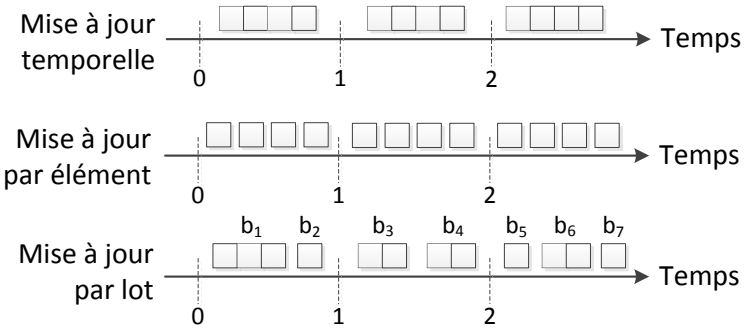


Figure 2.18 – Les différents modes de mise à jour d'une fenêtre.

fur et à mesure que la fenêtre « glisse » sur le flux. Enfin, les *fenêtres à repères* (*landmark window*) possèdent un point fixe et un point mobile ; elles sont dites *croissantes* (*growing*), lorsque leur point de départ est fixe, ou *décroissantes* (*contracting*), lorsque leur point d'arrivée est fixe (un point dans le futur, au moment où la fenêtre est créée).

Pour être cohérent, le comportement des fenêtres nécessite de spécifier (i) à quel moment ajouter les éléments reçus à la fenêtre (*mise à jour*) et (ii) quand est-ce que l'opérateur doit traiter l'ensemble fini d'éléments produit par la fenêtre (*déclenchement*) [130].

Comme montré sur la Figure 2.18 la mise à jour peut se faire au changement de timestamp (*time-driven*) ou à chaque nouvel élément (*tuple-driven*). Toutefois lorsque plusieurs éléments consécutifs ont le même timestamp, une fenêtre logique ignore quel est l'ordre des éléments à respecter. Pour résoudre ce problème, la solution proposée consiste à définir la relation d'ordre entre les éléments de même timestamp dès leur émission [131]. Dans ce troisième mode, l'émetteur spécifie lui même l'ordre des éléments de même timestamp, en créant des *lots*, et la fenêtre n'est mise à jour qu'à la fin d'un lot (*batch-driven*).

Selon le mode de déclenchement, les fenêtres peuvent porter un nom spécifique dans la littérature : une *fenêtre glissante* est déclenchée à chaque mise à jour, une *fenêtre sautante* (*jumping window*) [39] est déclenchée tous les k éléments ajoutés et une *fenêtre culbutante* (*tumbling window*) [132] est une fenêtre sautante où k est égal à la taille de la fenêtre.

2.2.1.3 État interne

Les différents opérateurs, qu'ils soient bloquants ou non bloquants, ont pour but d'appliquer des traitements sur des flux, avec ou sans l'application préalable d'une fenêtre. Pour pouvoir produire des résultats, ces opérateurs ont besoin d'un espace de travail où stocker, d'une part leurs paramètres, mais aussi les informations nécessaires à l'exécution du traitement au cours du temps. Dans ce dernier cas, on parlera alors d'*état interne* (*internal state*), c'est-à-dire l'ensemble des données qui doivent être conservées pendant toute l'exécution de l'opérateur.

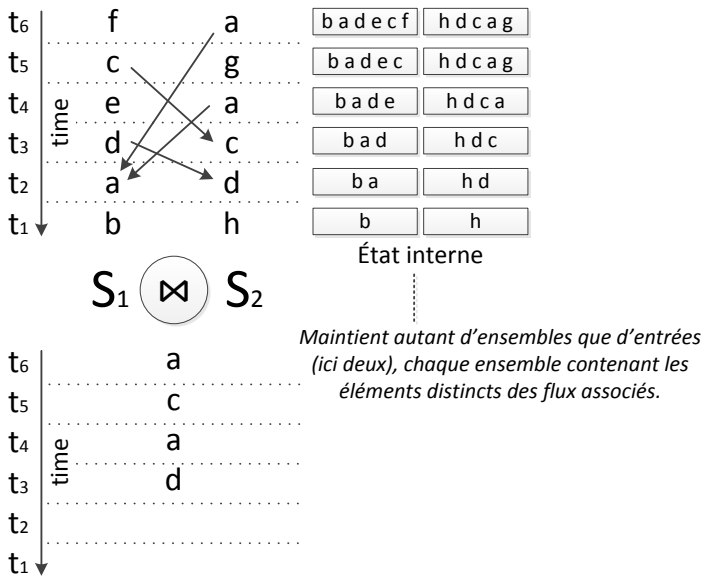


Figure 2.19 – Exemple de jointure continue.

Le comportement de l'état interne des opérateurs peut se caractériser au travers (i) de leur croissance en fonction de la taille des entrées et (ii) de leur capacité à être reconstruit à partir des flux d'entrée. L'évolution de la taille des opérateurs peut se décrire de la manière suivante [39] :

- Les *opérateurs sans état* (*stateless*), qui s'exécutent sans espace de travail. Par exemple, la projection et la sélection.
- Les *opérateurs à état constant* (ou *distributif*) qui possèdent un espace de travail de taille constante. Par exemple, le maximum et le comptage.
- Les *opérateurs holistiques*, dont l'état est proportionnel à la taille des entrées. Par exemple, la jointure, le comptage des éléments uniques [13] (*COUNT DISTINCT*) et l'intersection.

Pour plus de précision, on peut ensuite classer les opérateurs holistiques selon la croissance de leur espace de travail en fonction de la taille des entrées, comme cela se fait déjà le cadre de la complexité algorithmique : n (linéaire), n^k (polynomial), $n!$, k^n (exponentiel), $log(n)$ (logarithmique), $n \log(n)$, etc. Cette caractérisation permet de raisonner sur les prérequis nécessaires à l'utilisation des opérateurs, indépendamment de leur nature bloquante ou non. En effet, l'opération de jointure telle que décrite sur la Figure 2.19 possède un état qui croît linéairement en fonction de la taille des entrées (précisément, proportionnellement à $|S_1 \cup S_2|$). Bien que non bloquante, elle nécessite de fait l'utilisation de fenêtre pour pouvoir fixer le maximum de cet état.

Comme les opérateurs effectuent des traitements sur des flux, l'état interne est construit itérativement au fur et à mesure de la réception des n-uplets, sauf, bien entendu, si des fenêtres sont utilisées, auquel cas l'état interne est détruit à chaque évaluation

13. De manière générale, l'ensemble des opérations nécessitant d'éliminer les doublons.

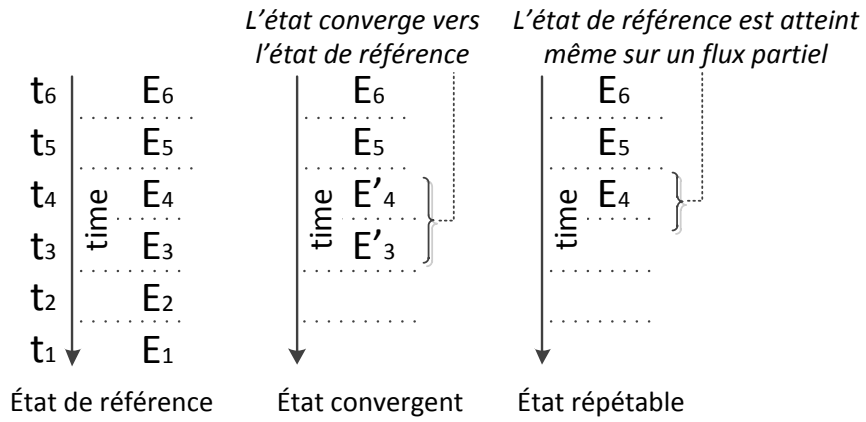
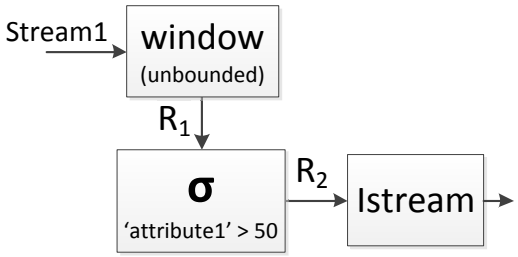


Figure 2.20 – Un état convergeant et un état répétable, comparés à leur état de référence.

du contenu de ces dernières. Le processus de construction de l'état peut être caractérisé au travers de sa capacité à être reconstruit à partir des flux d'entrée. Un opérateur est dit *déterministes* si, instancié plusieurs fois avec les mêmes flux en entrées, son état et sa sortie restent les mêmes à chaque élément traité. À l'inverse, un opérateur est dit *non déterministe* si plusieurs exécutions avec le même jeu de données n'aboutissent pas au même état. Trois grands types de traitements peuvent causer un comportement non déterministe [133] : (i) les traitements basés sur le temps courant (p. ex. les timestamps implicites ou l'horloge de la machine qui exécute l'opérateur), (ii) les traitements basés sur un générateur de nombres pseudo-aléatoires et (iii) les traitements basés sur l'ordre d'arrivée des n-uplets (typiquement, les opérateurs qui consomment plusieurs flux sont tributaires de la vitesse de production des émetteurs, mais aussi des canaux qui acheminement les flux). Dans le cas des nombres pseudoaléatoires, cependant, il est possible d'obtenir un comportement parfaitement déterministe à partir du moment où l'on considère l'état du générateur de nombres comme partie intégrante de l'état de l'opérateur.

Si un opérateur *déterministe* peut être reconstruit de manière incrémentale depuis le début de son exécution, en revanche le comportement de certains opérateurs permet une reconstruction plus rapide, qui ne nécessite pas forcément de traiter à nouveau l'ensemble du flux. On appelle *convergeant* un opérateur déterministe dont l'exécution à partir d'un point arbitraire du flux d'entrée converge à un moment donné vers le même état que celui obtenu par l'exécution sur le flux complet, comme l'illustre la Figure 2.20. Pendant un certain temps, appelé *instabilité*, l'état est différent, puis converge vers l'état de référence. Lorsqu'un opérateur convergeant ne possède pas de phase d'instabilité et peut être directement réobtenu depuis n'importe quel point du flux, il est alors dit *répétable*.

```
1 Select Istream(*)
2 From 'Stream1' [Range Unbounded]
3 Where 'attribute1' > 50
```



Cette requête se compose, tout d'abord, d'une fenêtre illimitée (ligne 2) qui construit une relation R_1 contenant tous les éléments du flux « Stream1 » jusqu'à l'instant présent. Cette relation est filtrée (ligne 3), produisant une autre relation R_2 à partir des éléments dont l'attribut « attribute1 » est supérieur à 50. Enfin, chaque nouvel élément de R_2 est écrit dans le flux de sortie (ligne 1).

Figure 2.21 – Un exemple de requête *CQL* et le plan de requête correspondant.

2.2.1.4 Déploiement et contrôle d'exécution

Tout comme dans un *DBMS*, les *DSMS* interprètent les requêtes déclaratives et construisent un *plan de requête* (*query plan*) qui définit quels seront les opérateurs à exécuter et dans quel ordre. En pratique, on trouve plusieurs types de plans spécifiques aux différentes phases d'analyse et d'exécution de la requête :

- Le *plan de requête logique*, qui correspond directement à la requête exprimée par l'utilisateur, comme montré sur la Figure 2.21. À noter que certaines approches, comme *Borealis* [134], proposent d'exprimer directement ce plan sans passer par une requête déclarative ; le plan est composé par l'utilisateur en manipulant des boîtes (les opérateurs) et des flèches (les flux) au travers d'un éditeur graphique.
- Le *plan de requête optimisé*, parfois appelé *plan de requête physique* [135], où les opérateurs peuvent être réordonnés, fusionnés ou remplacés par d'autres (optimisation statique). Cette phase d'optimisation peut aussi identifier les résultats intermédiaires pouvant être partagés avec des requêtes déjà en cours d'exécution dans le *DSMS*, de manière à éviter des redondances inutiles dans le calcul ou le stockage des données [39].
- Le *plan de distribution* [136], utilisé dans le cadre des *DSMS* distribués. À partir des informations sur le réseau, ce plan définit où les différents opérateurs doivent être placés de manière à satisfaire certaines contraintes (énergie, délai, etc.).

Le plan de distribution revêt une importance toute particulière dans le traitement distribué des flux, par exemple dans le cadre des réseaux de capteurs dont nous avons déjà parlé en première partie de ce chapitre. Dans l'ensemble, on trouve deux façons d'aborder la construction de ce plan : une *approche statique* et une *approche dynamique* [137].

Les approches statiques proposent de construire le plan de distribution dès la compilation de la requête déclarative et de produire le « meilleur » plan, c'est-à-dire celui qui minimise le coût d'exécution global exprimé par un modèle de calcul de coût pouvant

tenir compte de divers paramètres (coût énergétique, coût en mémoire, coût financier, coût en temps, etc.) en fonction du domaine. Concrètement, il s'agit de partitionner le plan de requête optimisé et d'associer chaque partition aux différents appareils du réseau en tenant compte notamment :

— des caractéristiques des appareils (énergie, ressources, etc.) ;

— des caractéristiques des opérateurs à déployer, tels que leur complexité algorithmique, leurs besoins en mémoire (état interne) ou leur ratio de débit (c.-à-d. le nombre moyen de n-uplets produits par l'opérateur à chaque n-uplet reçu en entrée, aussi appelé *sélectivité* de l'opérateur) ;

— des caractéristiques des flux d'entrée, notamment leur débit réel (p. ex. la fréquence d'échantillonnage) ou estimé (p. ex. la moyenne ou le maximum, mesurés empiriquement).

Outre le fait que ce partitionnement soit un calcul complexe en pratique [138], la nature statique de cette approche peut être sensible aux variations de charge, par exemple lorsque le débit d'un ou de plusieurs flux s'accroît ou lorsque le réseau est partagé entre plusieurs requêtes.

À l'inverse, les approches dynamiques suggèrent soit de (i) construire rapidement un plan de distribution initial qui sera raffiné ensuite au fur et à mesure de l'exécution [138], soit (ii) de ne pas utiliser de plan de distribution du tout et de disséminer directement le plan de requête optimisé dans le réseau, les différents appareils évaluant alors s'ils doivent y prendre part ou non [42]. Dans les deux cas, une fois le déploiement initial effectué, les opérateurs peuvent migrer d'un appareil à un autre ou se dupliquer lorsque cela s'avère nécessaire (accroissement de charge, batterie faible, etc.). Ce type d'approche induit un surcoût lié à l'analyse continue des propriétés des flux et de l'état des systèmes, ainsi qu'à la migration des opérations (tout particulièrement lorsque les états internes sont grands), mais celui-ci s'amortit avec le temps si les variations sont douces et que les périodes de stabilités sont longues [136]. Cependant, lorsque les variations sont nombreuses, rapides et courtes, le surcoût induit par des migrations répétées est prohibitif, conduisant à une saturation des appareils et du réseau de communication. Aussi, pour résoudre ce type de problèmes, une solution hybride a été proposée, où le plan de distribution initial est calculé spécifiquement pour assurer une certaine tolérance aux variations de charges de façon à limiter les migrations répétées, notamment en recherchant les configurations qui satisfont les plus grands écarts de charge possible [136, 137].

Une fois les opérateurs déployés et en cours d'exécution, l'acheminement des n-uplets doit être géré par le *DSMS*, qu'il s'agisse de la gestion des flux inter-opérateurs (*DSMS* distribué) ou de celle des flux d'entrée (tout *DSMS*), qui sont généralement acquis depuis des sources externes. Selon les caractéristiques du réseau (latence, pertes, etc.) et son état (congestionné, indisponible, etc.), certains n-uplets peuvent arriver dans le désordre ou avec un très fort retard. De la même façon, une source peut rester inactive pendant un certain temps, sans qu'il soit possible de déterminer si celle-ci a été déconnectée ou a

subi une panne, ou s'il n'y a simplement rien à transmettre. Aussi, pour contrôler à la fois l'exécution des opérateurs et les flots de données, les *DSMS* utilisent des n-uplets de contrôle (*control tuples*), directement insérés au sein des flux par les différents émetteurs. Ces n-uplets, appelés *ponctuations*, permettent au récepteur de recevoir des informations sur le flux qu'il consomme et d'utiliser ces informations pour optimiser son propre comportement [139].

À l'origine, une ponctuation est un prédicat garanti par l'émetteur pour le reste des n-uplets du flux de données ; par exemple, la garantie que toutes les valeurs qui vont suivre ont un timestamp supérieur à un seuil, et qu'en conséquence un n-uplet retardé doit être supprimé. Ainsi, tout élément futur qui ne respecterait pas les prédicats en cours de validité serait éliminé, car n'ayant plus d'importance pour le calcul en cours [140]. Les ponctuations peuvent être utilisées pour diverses optimisations, notamment pour éviter de traiter les n-uplets inutiles ou réduire la taille de l'état interne. Prenons l'exemple d'une jointure sur un attribut A du flux ; une ponctuation garantissant que la valeur de A sera supérieure à un seuil x nous permet de supprimer sans risque tous les n-uplets stockés dans l'état interne dont la valeur de A est inférieure ou égale à x [39]. Spécifiquement, les ponctuations dont le prédicat porte sur les timestamps sont appelés *battements de cœur*[14] (*heartbeats*) et sont utilisés, par exemple, pour (i) spécifier les intervalles de temps pendant lequel les éléments doivent être stockés dans un tampon avant d'être réordonnés et transmis à l'opérateur [141], ou pour (ii) indiquer explicitement la fin d'une fenêtre temporelle et transmettre immédiatement le contenu à un opérateur bloquant [142].

En pratique, lorsqu'un opérateur reçoit une ponctuation depuis un flux d'entrée S, celui-ci peut y réagir de manière spécifique [40] :

- dans le cas des opérateurs bloquants, en émettant mes résultats en attente (*comportement de publication*, ou *pass behavior*) ;
- dans le cas des opérateurs avec état, en nettoyant son état interne (*comportement de conservation*, ou *keep behavior*) ;
- en transformant la ponctuation et en la propageant sur sa sortie (*comportement de propagation*, ou *propagate behavior*).

Par exemple, un opérateur de tri bloquant pourra, à la réception d'une ponctuation, écrire sur sa sortie l'ensemble des éléments triés qui valident le prédicat associé. Une fois tous les éléments envoyés, cet opérateur pourra alors les retirer de son état interne puis propager immédiatement la ponctuation. Ces comportements peuvent être aussi définis par la sémantique des ponctuations utilisée. Par exemple, le prédicat temporel d'un *heartbeat* reste valide pour tous les descendants de l'opérateur dans le plan de requête et peut être propagé en toutes circonstances, après avoir été éventuellement mis à jour (modification du seuil temporel par rapport au temps courant) [140]. En outre, il a été

14. À ne pas confondre avec les protocoles de *heartbeat*, qui émettent des signaux périodiques (*heartbeats*) pour s'assurer que la communication entre deux machines est toujours ouverte. Lorsque le récepteur ne reçoit plus de *heartbeat* pendant un certain temps, la connexion est considérée comme perdue.

	***DSMS* relationnels**	***DSMS* génériques**
Traitement	Composition d'opérateurs prédéfinis (éventuel mécanisme d'extension)	Écrit par le développeur (éventuelle librairie d'opérateurs)
Expression	Langage de requêtes de haut niveau	Langage généraliste, langage dédié, agnostique (multi-langages)
Modèle de données	Bien défini	Pas de modèle de données
Flux	Structurés (schéma)	Potentiellement non structurés
Contrôle d'exécution*	Implicite, géré par le *DSMS*	Explicite, forte implication du développeur
Environnement d'exécution usuel	réseaux de capteurs, *clusters*, *cloud*	*clusters*, *cloud*

* Tolérance aux fautes, reprise sur incident, migration, redondance, synchronisation, etc.

Table 2.3 – Différences entre *DSMS* issu du modèle relationnel et *DSMS* générique.

suggéré d'utiliser des ponctuations *feedback*, qui circulent dans le sens inverse du flux (des récepteurs vers les émetteurs) pour contrôler le comportement des opérateurs en amont [143].

Au-delà des prédicats, une ponctuation désigne aujourd'hui d'autres types de n-uplets de contrôle. Par exemple, le contrôle d'accès et les politiques de sécurité peuvent être directement représentées sous la forme de ponctuations de sécurité qui permettent notamment de représenter des flux de données dont la criticité évolue au cours du temps. Ces ponctuations, lorsque reçue indiquent les politiques de sécurité que certains opérateurs doivent mettre en œuvre [144].

2.2.2 Plateformes génériques de traitement de flux

Dans les *DSMS* que nous avons présentés jusqu'à présent, le traitement se base principalement (i) sur des flux à la structure bien définie (les n-uplets et le schéma) et (ii) sur des requêtes continues exprimées dans des langages de haut niveau. Parallèlement, étant donné le besoin croissant de pouvoir développer des applications de traitement continu spécialisées, sur mesure et adaptées à des environnements spécifiques, d'autres types de systèmes de traitement de flux ont émergé : les *plateformes génériques de traitement de flux* (*general-purpose streaming platform*) [41]. Conçus pour pouvoir s'intégrer au sein des systèmes d'information existants et pour travailler avec des flux qui ne sont pas forcément structurés (sans schéma ou sous une forme purement binaire comme c'est le cas, par exemple, des flux audio et vidéo) issus de sources hétérogènes, ces *DSMS* proposent un cadre de développement doté [30] :

— d'un environnement d'exécution pour des applications de traitement continu et, la plupart du temps, distribué ;

— d'un environnement de développement proposant un ensemble de mécanismes bas niveau relatifs à la construction de ces applications, leur passage à l'échelle et le contrôle de leur exécution (tolérance aux fautes).

2.2.2.1 Traitement continu

Concrètement, les *DSMS* génériques sont, comme leur nom l'indique, plus généraux et moins structurés que les *DSMS* issus du modèle relationnel. Résumées dans la Table 2.3, les principales différences viennent du modèle de données et de la manière de décrire un traitement continu. Les *DSMS* dont nous avons parlé dans la section précédente proposent un modèle de données élaboré issu en grande partie du modèle relationnel, ce qui se traduit par la présence de schémas complexes définissant la structure des données et les contraintes sur leurs relations. Conformément à ce que l'on retrouve dans un *DBMS*, les traitements continus sont décrits comme des compositions d'opérateurs prédéfinis (p. ex. les opérateurs relationnels). Aussi, les *DSMS* relationnels tendent à se concentrer sur ces opérateurs, leur sémantique et leur optimisation. Au contraire, les *DSMS* génériques ne proposent pas de modèle de données particulier, ou un modèle très peu contraignant, et fournissent au développeur un cadre pour exprimer ses propres logiques de traitements sur les différents éléments du flux. En cela, ces *DSMS* se concentrent sur le modèle d'exécution, notamment dans des environnements de calcul parallèle et distribué (*clusters*, fermes de calculs, *cloud*, etc.) [30, 41]. À titre d'illustration, l'Extrait de code 2.22 montre comment un traitement continu est exprimé dans le *DSMS* généraliste *Apache Storm* [15]. On peut y observer les mécanismes de bas niveau utilisés pour extraire les valeurs depuis les n-uplets, écrire dans le flux de sortie et acquitter la bonne réception des données (tolérance aux fautes). Cependant, au-delà du cœur de l'approche, il n'est pas exclu qu'un *DSMS* relationnel propose un mécanisme d'extension permettant de créer des opérateurs personnalisés ou qu'un *DSMS* générique propose une bibliothèque d'opérateurs continus usuels [30] ; par exemple *IBM InfoSphere Streams* [16].

Cette différence d'approche a aussi un effet sur la façon d'exprimer les traitements continus. Un *DSMS* relationnel propose généralement un langage de requête de haut niveau pour effectuer la composition des opérateurs, tandis que les *DSMS* génériques vont reposer sur un ou plusieurs langages généralistes existants ou un langage dédié (*domain-specific language*, ou *DSL*). Par exemple, *Apache Storm* permet au développeur d'utiliser le langage de son choix, la plateforme d'exécution pouvant interagir avec les programmes qui traitent les données au moyen de messages échangés sur leurs interfaces d'entrée-sortie. De la même façon *InfoSphere Streams* se base sur un *DSL* de traitement de flux appelé *SPL* (*stream processing language*) et qui organise l'exécution d'opérateurs, ces derniers pouvant aussi être écrits dans les langages *Java* et *C++*.

15. *http://storm.incubator.apache.org* (accédé le 31/07/2014).

16. *http://www.ibm.com/software/products/en/infosphere-streams* (accédé le 31/07/2014).

```
1  public class DoubleAndTripleBolt extends BaseRichBolt
2  {
3      private OutputCollectorBase _collector;
4
5      @Override
6      public void prepare(Map conf, TopologyContext context, OutputCollectorBase collector)
7      {
8          _collector = collector;
9      }
10
11     @Override
12     public void execute(Tuple input)
13     {
14         int val = input.getInteger(0);
15         _collector.emit(input, new Values(val * 2, val * 3));
16         _collector.ack(input);
17     }
18
19     @Override
20     public void declareOutputFields(OutputFieldsDeclarer declarer)
21     {
22         declarer.declare(new Fields("double", "triple"));
23     }
24 }
```

Ici, nous étendons la classe de base pour créer une nouvelle opération continue (ou *bolt*, conformément au vocabulaire de *Storm*). Globalement, à chaque nouveau n-uplet reçu, la méthode *execute* est invoquée (ligne 11 à 17), ce qui provoque l'écriture du n-uplet résultat dans le flux de sortie (ligne 15). Un mécanisme d'acquittement manuel est aussi proposé par la plateforme, mis en œuvre à la ligne 16. Enfin, comme on peut le voir en ligne 22, chaque traitement doit déclarer une structure minimaliste (taille des n-uplets et noms des attributs) pour son flux de sortie.
Cet extrait de code provient de la documentation en ligne : *https://storm.incubator.apache.org/documentation/Tutorial.html* (accédé le 31/07/2014).

Figure 2.22 – Un traitement continu simple exprimé dans *Apache Storm*.

Étant typiquement conçus pour traiter de grands volumes de données au sein d'environnements de calcul parallèle et distribué, ces *DSMS* sont très souvent utilisés pour l'implémentation du niveau service des approches hiérarchiques (voir Chapitre 2), tout particulièrement dans le cadre des approches *cloud of Thing*. En outre, l'utilisation d'un *DSMS* générique possède trois grands avantages, du fait des libertés données au développeur par l'utilisation des mécanismes bas niveau. Tout d'abord, le développeur peut exprimer sa propre logique de traitement des n-uplets, sans aucune contrainte liée à l'expressivité du langage, ce qui fait des *DSMS* génériques les candidats idéaux pour traiter des flux de données non structurés. Ensuite, les bibliothèques et outils composant l'écosystème du langage généraliste (ou tout code métier antérieur) peuvent être directement réutilisés au sein des traitements continus. Enfin, le développeur peut exprimer des traitements adaptés à l'environnement de l'application et l'intégrer à des sources de données existantes (p. ex. un journal d'évènements, une base de données, un service Web, etc.), ce quel que soit le modèle de données employé. Par exemple, si les éléments

reçus sont compressés ou chiffrés, le développeur possède les moyens nécessaires pour exprimer la logique de décompression ou de déchiffrement.

Toutefois, si les *DSMS* génériques sont, par nature, plus extensibles que les *DSMS* issus du modèle relationnel, ils requièrent que le développeur s'intéresse à l'ensemble des problématiques bas niveau de la plateforme de traitement de flux. En effet, le développeur devient responsable de plusieurs aspects, tels que la structure et le format des n-uplets, l'optimisation des traitements ou la gestion des mécanismes de tolérance aux fautes (voir l'acquittement manuel dans l'Extrait de code 2.22).

En pratique, un *DSMS* générique se traduit sous la forme d'un intergiciel déployé sur les différentes machines qui vont composer l'environnement d'exécution des applications. Ces intergiciels possèdent deux fonctionnalités majeures [41] :

— permettre la communication entre les différentes entités qui produisent et consomment des flux (acheminement des n-uplets) et fournir les mécanismes nécessaires pour que ces échanges soient fiables et cohérents ;

— exécuter les traitements continus exprimés par le développeur, et éventuellement gérer leur cycle de vie et leur contexte d'exécution.

Si à l'origine les deux fonctions étaient fusionnées au sein d'un même produit, il est de plus en plus fréquent que l'infrastructure dédiée à l'exécution soit découplée de celle dédiée à la communication, donnant naissance à deux intergiciels distincts et, parfois, interchangeables [41].

2.2.2.2 Exécution des applications orientées flux

Le but d'un *DSMS* générique se résume à l'exécution d'applications spécifiques constituées de modules qui produisent et consomment des flux (*streaming applications*). Globalement, ces applications peuvent être exprimée au moyen de deux types de composants : (i) des sources de données, qui produisent des flux, et (ii) des composants de traitement, qui consomment et transforment les données produites par les sources pour produire de nouveau flux [41]. Ce type d'application peut se représenter naturellement sous la forme d'un graphe de flot de données (*data flow graphs*) [145], comme illustré sur la Figure 2.23. Ce graphe décrit comment les traitements s'enchaînent sur les éléments produits par les sources de données, et dans quel ordre : un arc du graphe représente une relation de précédence. Par exemple, sur la Figure 2.23, une exécution de P_2 est toujours précédée d'une exécution de P_1, elle-même toujours précédée de l'émission d'un n-uplet par S_1.

À partir de ce graphe, l'application est soumise à l'environnement d'exécution qui se compose d'un ensemble de machines connectées entre elles (un *cluster* typiquement). Conceptuellement, cet environnement se divise en plusieurs rôles qui sont assurés par les différentes machines : *nœud de calcul*, *coordinateur* et *interface de déploiement*.

Les nœuds de calcul participent à l'exécution de l'application et sont, de fait, les entités les plus répandues dans l'environnement d'exécution. En général, une opération décrite par le graphe de flot de données est encapsulée dans un composant logiciel appelé

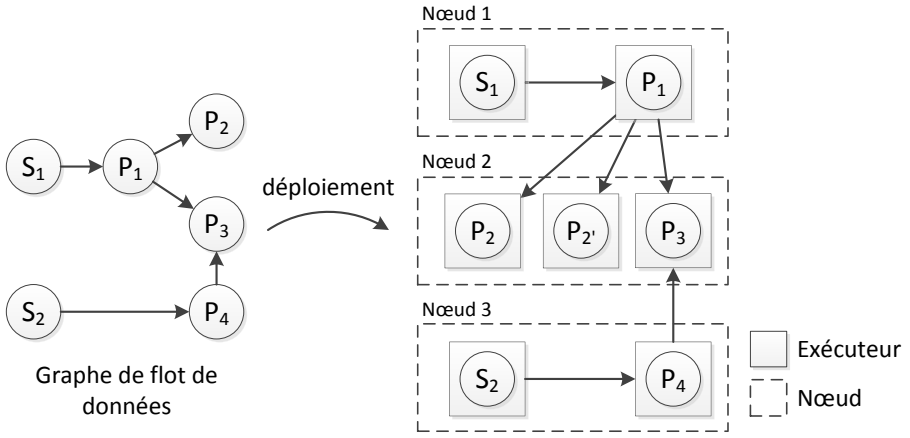


Figure 2.23 – Exemple de diagramme de flot de données, déployé dans un *cluster*.

exécuteur (ou *worker*) dont le rôle est de gérer l'exécution continue de l'opération (état interne, etc.) [41]. Chaque nœud de calcul est alors responsable du fonctionnement d'un ou de plusieurs exécuteurs exécutés en parallèle sur les différents cœurs disponibles (processeurs multicœurs). Une technique courante pour accroître la capacité des applications orientées flux à passer à l'échelle consiste à découper les flux de n-uplets en sous-flux [41]. Ainsi une opération peut être instanciée sous la forme de plusieurs exécuteurs répartis sur des nœuds ou des cœurs de calcul différents, comme montré sur la Figure 2.23 avec l'opération P_2 déployée en deux exemplaires sur le nœud 2. Typiquement, cette décomposition en sous-flux fait appel à un mécanisme permettant de déterminer quel n-uplet est affecté à chaque sous-flux. Cela peut consister à distribuer uniformément les n-uplets dans les sous-flux ou à utiliser des prédicats pour déterminer comment les n-upletsdoivent être groupés ; par exemple ceux dont la valeur d'un attribut est identique [30]. Ce découpage de flux est à rapprocher de la technique *MapReduce*, couramment utilisée pour le traitement de grands ensembles de données [146] et présentée dans l'Encart 2.3.

Les coordinateurs sont responsables du bon fonctionnement de l'environnement d'exécution et possèdent une vue globale du réseau et des applications qui s'y exécutent. Un coordinateur analyse en continu l'état des différents nœuds de calcul, soit en les contactant périodiquement pour obtenir des informations, soit en recevant des notifications de leur part. À partir des informations en sa possession, le coordinateur prend éventuellement des décisions destinées à répartir la charge (*load balancing*, migration des opérations) ou à garantir certaines propriétés de tolérance aux fautes : duplication préventive d'opérations, redémarrage des exécuteurs défaillants, etc. Bien que très souvent assuré par un coordinateur central, ce rôle peut impliquer plusieurs machines, chacune d'entre elles étant affectée à une portion spécifique de l'environnement d'exécution.

L'interface de déploiement a pour rôle de faire le lien entre l'environnement d'exécution et les utilisateurs de l'environnement d'exécution qui désirent déployer des applications. À la réception d'une application, l'interface de déploiement collabore avec les coordinateurs pour déterminer comment répartir les différentes opérations dans le

MapReduce est un paradigme de programmation pour l'écriture de traitements appliqués à de très grands ensembles de données, ces traitements étant parallélisés et distribués sur un *cluster* [146]. Pour ce faire, un programme *MapReduce* est exprimé en utilisant deux fonctions principales : *map*, qui crée des jeux de données intermédiaires, et *reduce*, qui traite les jeux de données intermédiaires. Par exemple, le comptage d'occurrence de chaque mot dans un ensemble de textes peut être implémenté sous la forme d'une fonction *map*, qui transforme les paires clé-valeur *(nom du document, contenu du document)* en ensembles de paires *(mot, 1)*, et d'une fonction *reduce* capable de traiter chaque mot séparément [146] :

```
map (nom, contenu):
    pour chaque mot dans contenu
        stocker la paire (mot, 1)
```

```
reduce (mot, paires):
    total = 0
    pour chaque (mot, nombre) dans paires
        total = total + nombre
    retourner total
```

Les systèmes qui implémentent *MapReduce* gèrent le stockage des jeu de données intermédiaires et dupliquent les fonctions *map* et *reduce* pour traiter les différentes parties de l'ensemble de données. Chaque instance de *map* et de *reduce* est alors affectée à un nœud et un cœur de calcul. Une troisième fonction est parfois utilisée, *merge*, pour agréger les résultats produits par les instances de *reduce* [146].

Encart 2.3 – La technique *MapReduce*.

réseau [30]. Par exemple, dans *Storm*, le graphe représentant l'application est construit par le développeur puis utilisé par l'interface de déploiement pour instancier les différents exécuteurs sur les nœuds de calculs. À l'inverse, *Samza* utilise un modèle de communication purement *publish-subscribe* où les flux sont représentés sous la forme de *topics*. Ainsi, il n'est pas nécessaire de connaître le graphe complet représentant l'application, chaque opération pouvant être déployée séparément en enregistrant ses exécuteurs à des *topics* existants et en créant de nouveaux *topics* correspondant aux nouveaux flux [41].

2.2.3 Traitement de flux dans le Web

Dans le Web, les flux de syndication sont spécifiquement conçus pour le suivi des modifications d'une ressource au cours du temps. En pratique, il s'agit pour les sites Web de mettre à disposition un simple fichier contenant une description, dans un format exploitable par les machines, des derniers changements apportés à une ressource donnée ; par exemple la liste des derniers articles publiés pour un journal en ligne. Lorsque l'on désire s'informer sur l'état de la ressource, il suffit alors de télécharger ce fichier et d'en extraire les informations désirées. Les *agrégateurs de flux* sont des logiciels qui récupèrent régulièrement ces fichiers et en présentent le contenu sous une forme graphique aux utilisateurs, les notifiant éventuellement des changements survenus entre deux téléchargements. Les deux technologies de flux de syndication utilisées sur le Web sont les flux *RSS* et les flux *Atom* [28], qui sont basés sur un format *XML* pour la description

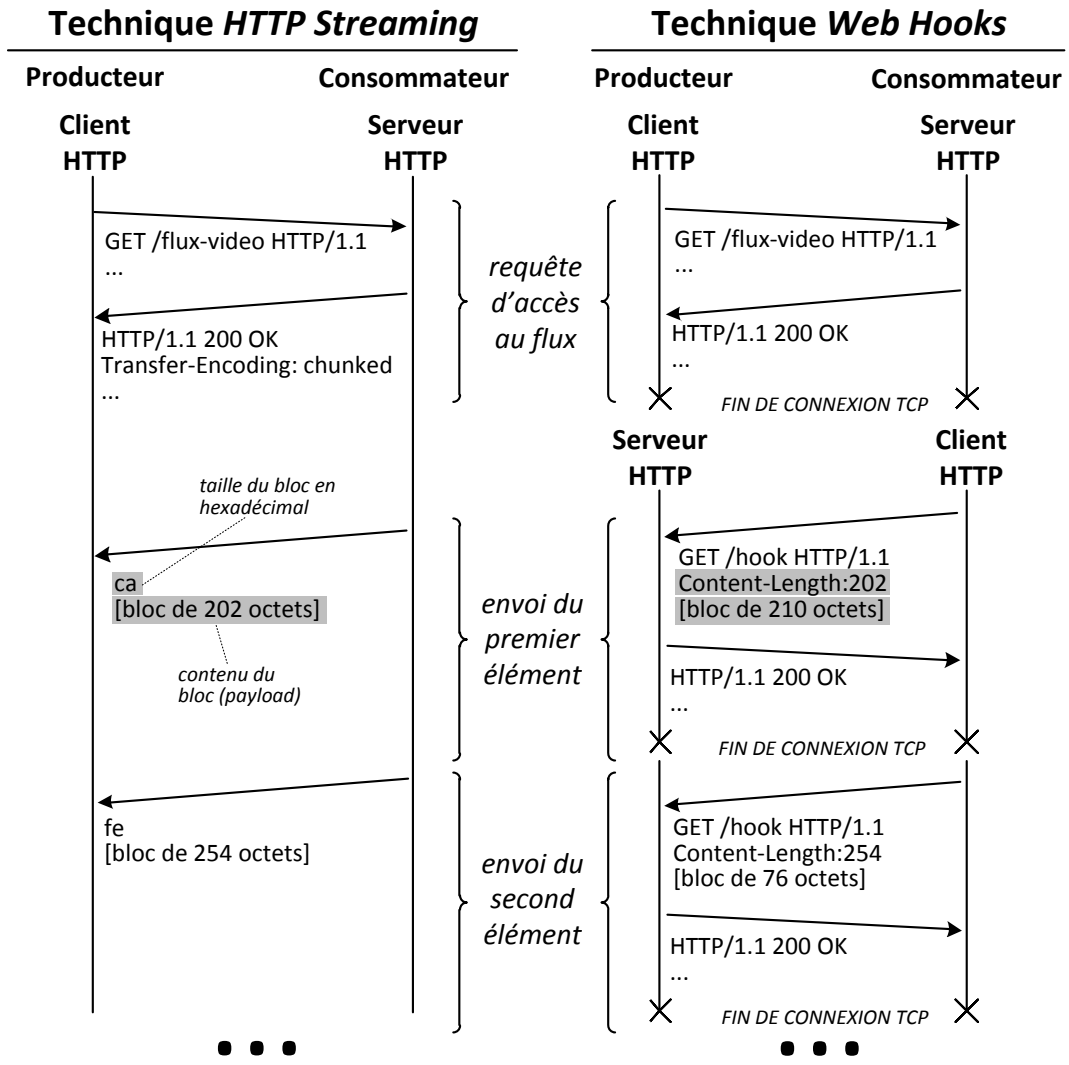


Figure 2.24 – Échanges réseaux pour *HTTP streaming* et *Web hooks*.

les dernières modifications d'une ressource. *Atom* présente l'avantage d'être étendu par le protocole *AtomPub* [147] qui permet notamment d'rajouter, de mettre à jour ou de supprimer des éléments dans un flux *Atom* au moyen de services *RESTful*. Si la possibilité d'utiliser les flux *Atom* a été suggérée pour la publication des données générées par des objets, certaines restrictions sont à considérer [148] : (i) les flux de syndication ne sont pas temps réel, (ii) l'expression de requêtes sur un flux n'est pas standardisée (iii) la verbosité de *XML* n'est pas adaptée aux réseaux bas débit, ce dernier point étant cependant sujet à discussion notamment du fait de l'existence du format *XML* compressé *EXI* [149].

Le transport de flux par *HTTP* est aujourd'hui très utilisé sur le Web, aussi bien pour des flux multimédias haut débit que pour des flux de notifications liés aux changements d'état d'une ressource. Une approche couramment utilisée pour les flux haut débit (nombre d'éléments par seconde élevé) est *HTTP streaming*, qui consiste à ne pas fermer la connexion *TCP* à la fin d'un cycle requête-réponse *HTTP* et à transmettre régulièrement les nouveaux éléments au fur et à mesure de leur production grâce au mode de transfert par bloc (*chunk*) du protocole *HTTP* [82]. La partie gauche de la Figure 2.24 illustre cette

technique par l'accès à un flux nommé */flux* et la transmission de deux n-uplets sous la forme de blocs de 202 et 254 octets respectivement. Pour les flux dont le débit est moindre, une autre technique est celle consistant à utiliser des *Web hooks*. La principale différence avec *HTTP streaming* réside dans le fait que le producteur et le consommateur embarquent tous les deux un client et un serveur *HTTP*. Le consommateur utilise son client *HTTP* pour contacter le serveur du producteur et s'enregistrer à un flux donné (*requête d'accès au flux*). Lors de cette requête, le consommateur indique au producteur un *URI* de retour (*callback URI*), qui sera invoqué par le producteur pour transmettre chaque nouvel élément produit. Cette méthode est illustrée par la partie droite de la Figure 2.24. Chacune des deux méthodes possède des avantages et des inconvénients propres. Les *Web hooks* introduisent un coût supplémentaire du fait des entêtes *HTTP* retransmis à chaque nouvel élément, là où *HTTP streaming* n'introduit quasiment aucun symbole supplémentaire hormis la taille des blocs. À l'inverse, *HTTP streaming* maintient la connexion *TCP* active, ce qui peut poser certains problèmes lorsque le nombre de connexions simultanées est limité par le système d'exploitation. De la même façon, maintenir une connexion *TCP* implique de conserver un ensemble d'informations en mémoire, le coût augmentant donc avec le nombre de flux servis en parallèle. De fait, *HTTP streaming* est plus adapté aux flux transportant beaucoup d'éléments par seconde, tandis qu'au contraire les *Web hooks* sont à privilégier lorsque les éléments sont très espacés dans le temps.

HTTP streaming est unidirectionnel, dans le sens où la réponse *HTTP* est un flot qui circule toujours du serveur vers le client récemment. Pour permettre des communications bidirectionnelles, l'*IETF* a standardisé le protocole *websockets* [150]. Celui-ci présente des performances similaires à *HTTP streaming* [151] mais rend possible la communication dans les deux sens pour les applications Web qui s'exécutent dynamiquement dans un navigateur. Le protocole *websocket* est composé en premier lieu d'une phase de négociation qui permet de faire la transition d'une connexion *HTTP* vers une connexion *websocket*. Une fois cette connexion mise en place, le client et le serveur peuvent échanger au moyen d'un protocole orienté message.

Au-delà du transport de flux directement sur *HTTP*, la problématique du transport de flux sur des messages *SOAP* a été envisagée dans le contexte des services *WS-**. La façon dont les messages *SOAP* sont échangés entre un fournisseur et un consommateur de service est contrôlée par un *schéma d'échange de message* (*message exchange pattern*, ou *MEP*) qui décrit notamment le cycle de vie des messages, leurs relations temporelles et les cas d'erreurs liés aux échanges [152]. Par défaut, la spécification *SOAP* définit uniquement des schémas d'échange pour des cas requête-réponse, notamment dans le cadre de l'invocation des services Web. Il a été suggéré de créer de nouveaux schémas d'échange pour supporter la diffusion de flux [153, 154], notamment en se basant sur *HTTP streaming* (requête unique et multiples réponses). Couplé à *EXI* pour la réduction de la taille des messages *SOAP*, l'écart de performance entre le transport de flux par messages *SOAP* ou par *HTTP streaming* devient négligeable [153]. En outre d'autres

schémas d'échange ont été proposés pour représenter des services qui consomment un flux en entrée et produisent un flux en sortie au fur et à mesure (multiples requêtes et multiples réponses) [153].

Enfin, un type de transport de flux peu étudié à l'heure actuelle est celui basé sur *CoAP*, étant donné que la spécification de ce protocole n'a été finalisée que récemment (juin 2014) [77]. Il est notamment possible d'implémenter un système de transport de flux inspiré des *Web hooks* directement dans *CoAP*, avec le bénéfice d'entêtes réduits à leur minimum. Cette technique est, par ailleurs, l'objet d'un brouillon de spécification (*draft*) pour l'implémentation *ressources observables* dans *CoAP* [155]. Cette spécification se base sur le patron de conception *Observateur* [156], bien connu dans le domaine du génie logiciel : des composants logiciels appelés *observateurs* s'enregistrent auprès d'un *sujet*, qui les notifie chaque fois que son état est modifié. De la même façon, un autre brouillon spécifie un équivalent du mode de transfert par bloc du protocole *HTTP* [157]. Ici, chaque bloc est transféré sous la forme d'une requête-réponse distincte ; les blocs étant généralement de taille fixe, négociée entre le client et le serveur. La réception de chaque bloc est acquittée séparément, *CoAP* prenant en charge la réémission des paquets perdus si nécessaire. En outre, la diffusion de blocs peut se faire aussi bien du client vers le serveur (envoi de flux) que du serveur vers le client (réception de flux).

2.3 Conclusion sur le traitement de flux dans l'Internet des objets

De notre état de l'art, il ressort que les solutions pour l'Internet des objets sont très hétérogènes, du fait des différentes visions impliquées et qui se concentrent d'un côté sur les objets physiques (*RFID* et réseaux de capteurs) et de l'autre sur le réseau Internet et le Web. Par exemple, nous voyons que pour le cas du traitement des flux de données, lorsqu'il est abordé dans le cadre de l'Internet des objets, n'est pas géré de la même façon selon les visions. Les visions qui se concentrent sur les objets physiques ont introduit un ensemble de *DSMS* essentiellement relationnels là où la vision Web introduit des approches basées sur des systèmes de traitement de flux génériques. Cela peut s'expliquer notamment par la hiérarchisation de l'Internet des objets adopté pour la gestion de la très grande échelle. Les approches orientées Internet auront en effet tendance à se situer au niveau des services tiers, tout particulièrement dans le cadre du Web des objets. À l'inverse, les approches orientées objet auront tendance à privilégier des approches de type *in-network processing*, où la logique de traitement et d'action est répartie au sein du réseau d'objets.

À l'heure actuelle, l'étude du traitement continu et de la conception d'applications orientées flux dans l'Internet des objets n'est que très marginalement représentée, notamment en ce qui concerne la conception d'applications, leur déploiement et leur distribution dans les réseaux d'objets. À notre connaissance les architectures orientées service, qu'elles

soient spécifiques à l'Internet des objets ou non, ne questionnent pas le problème du traitement continu et des flux de données. De la même façon se pose la question de la représentation et du développement des applications orientées vers le traitement continu de flux de données au sein de l'Internet des objets. Si les différents travaux se concentrent sur les services tiers ou sur les objets eux-mêmes, il est nécessaire d'introduire une solution approchant la structure hiérarchique de l'Internet des objets de manière globale, conformément aux besoins des applications et aux caractéristiques de ces différents niveaux.

Chapitre 3

Modèle de données et de calcul orienté flux pour l'Internet des objets

---

Notre approche repose sur un cadre théorique homogène pour le traitement continu, adapté aux particularités de l'Internet des objets et issu de l'analyse de l'état de l'art que nous avons présenté au chapitre précédent. Notre modèle de données permet de décrire des flux de données structurées qui évoluent au cours du temps et notre modèle de calcul introduit une architecture orientée service pour la conception d'applications qui produisent, transforment, stockent et consomment des flux.

I	Élément de flux.
$I[k]$	Valeur du k-ème attribut de I ou valeur de l'attribut nommé k de I.
S, $\|S\|$	Flux et taille du flux.
$\mathcal{I}$	Suite d'éléments composant un flux.
I_i	i-ième élément d'une suite d'éléments $\mathcal{I}$.
$I \in S$	Appartenance d'un élément I à la suite d'éléments $\mathcal{I}$ d'un flux S.
$\mathbb{I}$	Support de flux.
$\mathcal{S}$	Schéma de flux.
$\rho(t), \rho$	Fonction de débit et débit moyen d'un flux.
$\mathcal{T}$	Fonction d'association des valeurs de temps aux éléments d'un flux.
$\mathbb{T}$	Ensemble des valeurs de temps possibles.
$\mathbb{K}$	Ensemble des identifiants.
Σ	État interne d'un transformateur.
$\Sigma[k]$	Valeur du k-ème attribut de Σ ou valeur de l'attribut nommé k de Σ.
σ, σ^+	Fonction de construction et fonction de croissance de Σ.
$\mathcal{R}$	Ensemble ordonné exprimant un ordre supplémentaire sur un flux.
$\dot{p}$	Présent d'un ordre.
$S_1 \overset{\mathcal{S}}{\sim} S_2$	S_1 et S_2 sont s-communs.
$S_1 \sim S_2$	S_1 est similaire à S_2.
$S_1 \subset S_2$	S_1 est un sous-flux de S_2.
$S_1 \overset{\leq}{\subset} S_2$	S_1 est un sous-flux ordonné de S_2.
$S_1 \sqsubset S_2$	S_1 est un sous-flux continu de S_2.
S^k	Portion de S constituée des k premiers éléments.
$S^{a \to a}$	Portion de S constituée des éléments I_a à I_b.
$\mathcal{S}_1 \subset \mathcal{S}_2$	$\mathcal{S}_1$ est un sous-schéma de $\mathcal{S}_2$.
$S_1 =_{a_1,\cdots,a_n} S_2$	S_1 est un flux partiel de S_2 par rapport aux attributs $a_1, \cdots, a_n$.
$S_1 \subset_{a_1,\cdots,a_n} S_2$	S_1 est un sous-flux partiel de S_2 par rapport aux attributs $a_1, \cdots, a_n$.
$S_1 \overset{\leq}{\subset}_{a_1,\cdots,a_n} S_2$	S_1 est un sous-flux partiel ordonné de S_2 par rapport aux attributs $a_1, \cdots, a_n$.
$S_1 \sqsubset_{a_1,\cdots,a_n} S_2$	S_1 est un sous-flux partiel continu de S_2 par rapport aux attributs $a_1, \cdots, a_n$.
$\|$	Opérateur de concaténation.
$\alpha, \beta, \gamma, \delta$	Prédicat de mise à jour, fonction d'évaluation, fonction de transformation et prédicat de déclenchement d'une fenêtre.
w_i	État d'une fenêtre.
$\|w_i\|$	Taille physique d'une fenêtre.
$\|W\|_{max}$	Taille physique maximale d'une fenêtre.
p_s, p_e	Borne inférieure et borne supérieure d'une fenêtre.
W_φ	Fenêtre positionnelle.
W_θ	Fenêtre temporelle.
$\triangleright$	Fonction de construction d'ordre d'un *streamer*.

Table 3.1 – Récapitulatif des notations utilisées pour le modèle de données.

Un *multiensemble* est une généralisation de la notion d'ensemble, où un même élément peut apparaître plusieurs fois. Un multiensemble, noté (A, m), se compose d'un ensemble A, dit *support*, et d'une *fonction de multiplicité* m qui retourne le nombre d'occurrences d'un élément x dans le multiensemble ; $m(x) = 0$ indiquant que l'élément x n'est pas présent dans A. Une autre notation consiste à écrire $\{\{a, a, b, c, c, c\}\}$ le multiensemble $(\{a, b, c\}, m)$ où $m(a) = 2, m(b) = 1, m(c) = 3$ et $m(x \notin \{a, b, c\}) = 0$.

Encart 3.1 – Rappel sur le vocabulaire relatif aux multiensembles.

Ce chapitre aborde la problématique de représentation et de traitement des flux au travers d'un cadre théorique uniforme, reformulant et complétant les concepts introduits dans la littérature pour les différents besoins des systèmes de traitement de flux pour réseaux de capteurs. Ce travail s'articule autour d'un modèle de données pour les flux et un modèle de calcul pour le traitement continu, introduisant une architecture orientée service pour la conception des applications orientées flux qui s'exécutent dans l'Internet des objets.

3.1 Modèle de données pour l'Internet des objets : flux continu

Notre modèle de données décrit comment les informations sont structurées au sein d'un flux de données et comment celles-ci circulent d'un émetteur vers un récepteur : schéma de données, relations temporelles, caractéristiques des flux.

3.1.1 Flux, élément de flux, schéma de flux

De manière analogue aux différentes approches de traitement de flux, la plus petite unité d'information représentable dans notre approche est l'*élément de flux* (*stream item*). Ces éléments circulent entre un émetteur et un récepteur, formant un flux :

Définition 3.1 Élément de flux Un *élément de flux*, ou *élément*, est un n-uplet I représentant une information circulant dans un flux. La notation $I[k]$ peut désigner soit (i) la valeur du k-ème attribut du n-uplet I, si $k \in \mathbb{N}^*$, soit (ii) la valeur de l'attribut k du n-uplet I, si k est un nom d'attribut.

Définition 3.2 Flux Un *flux* S est une suite d'éléments de flux $\mathcal{I} = (I_i)_{1 \leq i \leq |S|}$ dont la longueur est notée $|S|$ et peut être infinie.

Définition 3.3 Support de flux Étant donné qu'un même élément I peut apparaître plusieurs fois dans un flux S avec un indice différent, on appelle *support du flux* S, noté $\mathbb{I}$, l'ensemble des éléments distincts pouvant apparaître dans S. Par souci de concision, on note directement $I \in S$ l'appartenance d'un élément I à l'ensemble $\mathbb{I}$ d'un flux S.

Par souci de concision, nous utiliserons, lorsque cela s'y prête, des notations usuelles différentes pour décrire la suite $\mathcal{I}$ des éléments du flux S :

- La *notation fonctionnelle* : la suite d'éléments est décrite par une fonction $\mathcal{I} : [\![1, |S|]\!] \longrightarrow \mathbb{I}$, associant à chaque indice i son élément correspondant dans le flux.
- La *notation ensembliste* : la suite d'éléments est représentée par un ensemble de couples élément-indice $\mathcal{I} = \{(I_i, i)\}_{1 \leq i \leq |S|}$, totalement ordonné par la relation d'ordre définie sur les indices : $\forall (I_i, i), (I_j, j) \in \mathcal{I}, (I_i, i) \leq (I_j, j) \Leftrightarrow i \leq j$. On pourra y adjoindre une fonction de multiplicité, qui à tout élément $I \in \mathbb{I}$ associe le nombre d'apparitions de I dans la suite.
- La *notation multiensembliste* : lorsque l'ordre des éléments n'a pas d'importance, la suite d'éléments est représentée par un multiensemble dont le support est $\mathbb{I}$ et dont la fonction de multiplicité retourne le nombre d'occurrences d'un élément $I \in \mathbb{I}$ dans la suite $\mathcal{I}$.

Dans le cas usuel, chaque élément I de S respecte une même structure, décrite par un *schéma de flux*, ou *schéma* :

Définition 3.4 Schéma de flux Le schéma d'un flux S, noté $\mathcal{S}$, décrit les propriétés relatives à la structure de tous les éléments de $I \in S$, c'est-à-dire (i) la taille des éléments et (ii) les métadonnées associées à chaque attribut : nom de l'attribut, position de l'attribut dans les éléments, domaine de l'attribut, type de l'attribut.

Définition 3.5 Domaine d'un attribut Le domaine d'un attribut x est l'ensemble des valeurs possibles prises par x dans tous les éléments du flux.

Définition 3.6 Type d'un attribut Le type d'un attribut est caractérisé par un *type sémantique*, un *type concret* et une *unité*. Le type sémantique indique quelle est la nature des valeurs de l'attribut, le type concret indique quel est l'encodage des valeurs de l'attribut et l'unité spécifie dans quelle unité sont exprimées les valeurs de l'attribut.

Typiquement, les unités sont celles définies par le système international [158], que nous complétons par (i) les unités monétaires, (ii) une unité *pourcent* (%) et (iii) la possibilité de définir un attribut comme étant sans unité. Le type sémantique fait référence à des modèles de données existants, tels que les ontologies de capteurs et de concepts physiques que nous avons présentés au Chapitre 2, ou peut être personnalisé pour des usages spécifiques. De la même façon qu'une information peut n'avoir aucune unité, une information peut n'avoir aucun type sémantique particulier. En pratique, le type sémantique et l'unité sont exploités pour raisonner à propos des éléments du flux. Par exemple, étant donné un flux de valeurs de température exprimées en degrés Celsius, la conversion vers une autre unité (p. ex. conversion en kelvin) est réalisée directement à partir des formules exprimées dans les ontologies associées. De manière analogue, composer les valeurs de plusieurs capteurs permet d'obtenir de nouvelles informations qui ne seraient pas fournies par un capteur unique ; par exemple, le refroidissement éolien

(indice censé représenter la température ressentie, calculé à partir de la température et de la vitesse du vent) ou l'indice de chaleur (idem, mais basé sur la température et l'humidité relative).

Le type concret d'un attribut indique comment les différentes informations peuvent être décodées. Les types suivants forment la base des types concrets : *réel*, *entier*, *booléen*, *chaîne de caractères* et *séquence d'octets* (*blob*). Pour représenter des types concrets plus complexes, tels que des images, des données audio ou des données vidéo, les *types MIME*[1] sont utilisés.

3.1.2 Familles de flux et flux remarquables

En considérant tout ce que nous venons de poser, un flux S est noté, dans sa forme générale, par $S = (\mathcal{I}, \mathcal{S})$. Un flux représente, au travers de la suite d'éléments $\mathcal{I}$, n'importe quel jeu de données que l'on souhaite traiter de manière continue. Toutefois, les flux peuvent posséder des caractéristiques additionnelles, définies comme suit :

Définition 3.7 Flux non structuré Un flux est dit *non structuré* s'il ne possède pas de schéma, noté $S = (\mathcal{I}, \varnothing)$, où chaque élément I possède sa propre structure.

Définition 3.8 Flux temporel Un flux est dit *temporel*, noté $S = (\mathcal{I}, \mathcal{S}, \mathcal{T})$, s'il est doté d'une fonction $\mathcal{T}$ associant une information temporelle à chaque élément I de S.

Les différentes valeurs de temps possibles forment un ensemble totalement ordonné, noté $\mathbb{T}$, en théorie isomorphe à $\mathbb{R}^+$ (temps continu). En pratique, le temps est souvent représenté de manière discrète, sous la forme d'une partie de l'ensemble $\mathbb{N}$ (timestamps). En conséquence, plusieurs éléments du flux peuvent avoir une même valeur de temps : dans ce cas de figure, $\mathbb{T}$ est seulement ordonné.

Définition 3.9 Flux identifié Un flux est dit *identifié*, noté $S = (\mathcal{I}, \mathcal{S}, \mathfrak{I})$, si son schéma définit l'un des attributs comme étant un identifiant unique et définitif pour chaque élément I de S. Par définition, les éléments de la suite $\mathcal{I}$ d'un flux identifié sont uniques. On note $\mathfrak{I}$ la fonction qui associe la valeur de son identifiant à un élément I.

Les différentes valeurs d'identifiants forment un ensemble totalement ordonné, noté $\mathbb{K}$ qui est typiquement isomorphe à $\mathbb{N}*$.

Définition 3.10 Flux t-identifié Un flux temporel est dit *temporellement identifié*, ou *t-identifié*, lorsque la fonction $\mathcal{T}$ est injective, c'est-à-dire n'associe jamais à deux éléments la même valeur de temps : $\mathbb{K} = \mathcal{T}$.

1. Un *Internet media type*, ou *type MIME*, est un nom en deux parties utilisé pour identifier un format de donnée spécifique ; par exemple, « image/jpeg » pour le format d'image *JPEG*. Ces types étaient à l'origine utilisés dans le protocole *SMTP* [159], mais ont été notamment étendus à *HTTP* pour spécifier le format des données contenues dans un corps de requête ou de réponse.

Définition 3.11 Flux historique Un flux est dit *historique*, noté $S = (\mathcal{I}, \mathcal{S}, H)$, s'il est possible de lire à nouveau tout ou partie des éléments produits par le passé. L'historique associé au flux correspond à l'ensemble des éléments de $\mathcal{I}$ qui satisfont le prédicat H.

Le plus simple historique d'un flux S est la sous-suite continue de $\mathcal{I} : (I_i, \cdots, I_j)_{0 \leq i < j \leq |S|}$. Pour cet historique, le prédicat H est vérifié pour un élément $I_i \in \mathcal{I}$ lorsque $i \geq i \leq j$.

3.1.3 Notion d'ordre

L'ordre des éléments dans un flux revêt une importance fondamentale, en cela qu'il influe sur les résultats produits par les différents traitements continus appliqués au flux. Par exemple, considérons un traitement qui, à partir d'un flux S de 1-uplets, produit un flux S' dans lequel tous les éléments dont l'attribut a est inférieur à un seuil k sont retirés : $\mathcal{I}' = (I_i \in S | I_i[a] > k)$. Étant donné que le traitement est appliqué sur chaque élément successivement, l'ordre des éléments de S influe naturellement sur l'ordre des éléments de S'.

L'ordre des éléments dans le flux est naturellement positionnel puisque défini par les indices de $\mathcal{I}$:

Définition 3.12 Ordre positionnel L'*ordre positionnel* d'un flux S, ou *ordre naturel*, correspond à la relation d'ordre total établie sur l'ensemble des indices de la suite $\mathcal{I} = (I_1, \cdots, I_n)$, cet ensemble étant inclus dans l'ensemble des entiers strictement positifs $\mathbb{N}^* : \forall x, y \in \mathbb{N}^*, I_x \leq I_y \Rightarrow x \leq y$

L'ordre positionnel est toujours celui fixé par la source de données qui produit le flux ; il est donc toujours défini sur le flux. Cependant, un flux peut être associé à plusieurs relations d'ordre, dîtes *ordres supplémentaires*.

Définition 3.13 Ordre supplémentaire Un *ordre supplémentaire* est une relation d'ordre définie sur les éléments d'un flux S et qui respecte l'ordre naturel. Soit un flux S et $\mathcal{R}$ un ensemble totalement ordonné, on dira que S est *ordonné par* $\mathcal{R}$ s'il existe une application injective $f : [\![1, |S|]\!] \longrightarrow \mathcal{R}$ qui respecte la croissance des indices :

- S *est ordonné par* $\mathcal{R}$ si $\forall i, j \in [\![1, |S|]\!], i < j \Rightarrow f(i) \leq f(j)$.
- S *est totalement ordonné par* $\mathcal{R}$ si $\forall i, j \in [\![1, |S|]\!], i < j \Rightarrow f(i) < f(j)$.

Si n'importe quel ensemble totalement ordonné peut être utilisé pour ordonner un flux, nous pouvons cependant décrire sous la forme d'ordres supplémentaires les ordres usuels introduits par la littérature sur le traitement de flux :

Définition 3.14 Ordre temporel L'*ordre temporel* d'un flux temporel S est défini sur l'ensemble $\mathbb{T}$ des valeurs de temps. L'ensemble $\mathbb{T}$ peut ordonner S de manière totale ou non.

Soit deux ensembles A et B ordonnés partiellement, c'est-à-dire possédant une relation d'ordre définie sur certains couples d'éléments. L'*ordre lexicographique* sur le produit cartésien $A \times B$ est une relation d'ordre définie par :

$$\forall (a, b), (a', b') \in A \times B$$

$$(a, b) \leq (a', b') \Rightarrow \begin{cases} \text{soit } a < a', \\ \text{soit } a = a' \text{ et } b \leq b'. \end{cases}$$

Si A et B sont totalement ordonnés, alors $A \times B$ est lui aussi totalement ordonné par la relation d'ordre lexicographique.

Encart 3.2 – Ordre lexicographique sur produit cartésien.

Définition 3.15 Ordre des identifiants L'*ordre des identifiants* d'un flux identifié S est défini sur l'ensemble $\mathbb{K}$ des valeurs d'identifiants. L'ensemble $\mathbb{K}$ ordonne totalement S.

Un troisième ordre usuel apparaît dans la littérature pour lever l'ambiguïté posée par le cas où plusieurs éléments d'un flux temporel ont un même timestamp. Pour ce faire, les éléments sont classés par *lots* :

Définition 3.16 Ordre lot-temps L'*ordre lot-temps* d'un flux temporel $S = (\mathcal{I}, \mathcal{S}, \mathcal{T})$ est un ordre lexicographique (voir Encart 3.2) défini sur l'ensemble $\mathbb{N}^* \times \mathbb{T}$. L'ensemble $\mathbb{N}^*$ représente l'ordre des lots, un lot étant une sous-suite continue $\mathcal{I}'$ de $\mathcal{I}$ pour laquelle le flux $S' = (\mathcal{I}', \mathcal{S}, \mathcal{T})$ est totalement ordonné par $\mathbb{T}$. L'ensemble $\mathbb{N}^* \times \mathbb{T}$ ordonne totalement S.

Le concept d'ordre étant posé, nous devons formaliser la notion de *présent*, étant donné que nous sommes amenés à manipuler les éléments d'un flux au fur et à mesure de leur production :

Définition 3.17 Présent d'un ordre Le *présent d'un ordre* défini sur $\mathcal{R}$ est une variable $\dot{p} \in \mathcal{R}$ dont la valeur évolue continuellement selon une définition arbitraire :

- pour l'ordre positionnel, il s'agit de l'indice du dernier élément manipulé $x \in \mathbb{N}^*$;
- pour l'ordre temporel, il s'agit du temps courant $x \in \mathbb{T}$;
- pour l'ordre lot-temps, il s'agit du couple $(x, y) \in \mathbb{N}^* \times \mathbb{T}$, avec x le numéro du dernier lot connu et y le temps courant.

3.1.4 Relations structurelles entre les flux

Conformément à la relation d'ordre qui régit les indices, il est possible d'exprimer l'indice d'un élément I à partir du nombre d'éléments qui précédent I dans le flux. De

là, nous définissons les relations structurelles qui peuvent exister entre les flux, la plus simple étant l'égalité :

Définition 3.18 Égalité de flux Deux flux $S_1 = (\mathcal{I}_1, \mathcal{S}_1)$ et $S_2 = (\mathcal{I}_2, \mathcal{S}_2)$ sont *égaux*, ou *identiques*, si et seulement si $\mathcal{I}_1 = \mathcal{I}_2$ et $\mathcal{S}_1 = \mathcal{S}_2$.

Si l'égalité entre deux suites $\mathcal{I}_1$ et $\mathcal{I}_2$ est mathématiquement définie, l'égalité entre schémas a pour particularité de ne pas forcément respecter l'ordre des attributs de $\mathcal{S}_1$ et $\mathcal{S}_2$:

Définition 3.19 Égalité de schéma Deux schémas sont dits *égaux* si $\mathcal{S}_1 \cap \mathcal{S}_2 = \mathcal{S}_1 \cup \mathcal{S}_2$ et *parfaitement égaux* si l'ordre des attributs est respecté : $\exists a_i \in \mathcal{S}_1 \Leftrightarrow \exists a_j \in \mathcal{S}_2$ tel que $i = j$.

Définition 3.20 Flux schéma-communs Deux flux sont dits *schéma-communs*, ou *s-communs*, si leurs schémas sont identiques, deux flux identiques étant naturellement s-communs. Cette relation est notée $S_1 \overset{\mathcal{S}}{\sim} S_2$.

Définition 3.21 Flux similaires Un flux S_1 est dit *similaire* à un flux S_2, noté $S_1 \sim S_2$, si la suite $\mathcal{I}_1$ contient les mêmes éléments que $\mathcal{I}_2$, en même quantité, mais dans un ordre différent :

$$S_1 \sim S_2 \Rightarrow \begin{cases} S_1 \text{ et } S_2 \text{ sont s-communs,} \\ |S_1| = |S_2|, \\ \forall I_i \in \mathcal{I}_1, \exists I_j \in \mathcal{I}_2 \text{ tel que } I_i = I_j. \end{cases}$$

Définition 3.22 Sous-flux Un flux S_1 est dit *sous-flux* de S_2, noté $S_1 \subset S_2$, si chaque élément de S_1 se retrouve à l'identique dans S_2 à la même position ou à une position différente. Cette relation est proche de la similarité, mais avec des flux de tailles différentes :

$$S_1 \subset S_2 \Rightarrow \begin{cases} S_1 \text{ et } S_2 \text{ sont s-communs,} \\ |S_1| < |S_2|, \\ \forall I_i \in \mathcal{I}_1, \exists I_j \in \mathcal{I}_2 \text{ tel que } I_i = I_j. \end{cases}$$

Autrement dit, S_1 est un sous-flux de S_2 s'il existe une application injective qui, à tout indice de $\mathcal{I}_1$, associe un seul indice de $\mathcal{I}_2$:

$$S_1 \subset S_2 \Rightarrow \exists f : [\![1, |S_1|]\!] \longrightarrow [\![1, |S_2|]\!] \text{ tel que } \forall I_i \in S_1, I_i = I_{f(i)} \in S_2$$

Définition 3.23 Sous-flux ordonné Un flux S_1 est dit *sous-flux ordonné* de S_2, noté $S_1 \overset{\leq}{\subset} S_2$, si l'application injective f préserve la relation d'ordre des indices initiaux :

$$S_1 \overset{\leq}{\subset} S_2 \Rightarrow \forall i, j \in [\![1, |S_1|]\!], i \leq j \Leftrightarrow f(i) \leq f(j)$$

Définition 3.24 Sous-flux continu Un flux S_1 est dit *sous-flux continu*, ou *portion*, de S_2 si la suite $\mathcal{I}_1$ se retrouve exactement dans $\mathcal{I}_2$, avec des indices identiques à une constante près :

$$S_1 \sqsubset S_2 \Rightarrow \begin{cases} S_1 \text{ est un sous-flux ordonné de } S_2, \\ \exists k \in \mathbb{N}, \text{ tel que } \forall i \in [\![1, |S_1|]\!] \text{ on a } f(i) = k + i \end{cases}$$

Par souci de simplification, on notera S^i le sous-flux continu de S contenant les i premiers éléments du flux, c'est-à-dire la suite $\mathcal{I} = (I_k)_{1 \leq k \leq i}$. Par extension, on notera $S^{i \to j}$, avec $i, j \in [\![1, |S|]\!]$ et $i < j$, le sous-flux continu constitué des éléments i à j : $\mathcal{I} = (I_k)_{i \leq k \leq j}$.

Les relations qui existent entre les sous-flux et les super-flux peuvent être complétées par les relations qui existent entre les schémas :

Définition 3.25 Sous-schéma Un schéma $\mathcal{S}_1$ est dit *sous-schéma*, ou *partie*, d'un schéma $\mathcal{S}_2$ si les différents attributs de $\mathcal{S}_1$ se retrouvent dans $\mathcal{S}_2$:

$$\mathcal{S}_1 \subset \mathcal{S}_2 \Rightarrow \begin{cases} |\mathcal{S}_1| < |\mathcal{S}_2|, \\ \forall x_i \in \mathcal{S}_1, \exists y_j \in \mathcal{S}_2 \text{ tel que } x_i = y_j \end{cases}$$

Définition 3.26 Flux partiel Un flux S_1 est dit *flux partiel* de S_2 si le schéma de S_1 est une partie de S_2 et si S_1 est égal à S_2 lorsque l'on ne tient compte que de la partie du schéma commune aux deux flux. Formellement, soient $a_1, \cdots, a_n$ les noms d'attributs communs à S_1 et S_2 :

$$S_1 =_{a_1,\cdots,a_n} S_2 \Rightarrow \begin{cases} \mathcal{S}_1 \subset \mathcal{S}_2, \\ \forall I_i \in \mathcal{I}_1, \exists I_j \in \mathcal{I}_2 \text{ tel que } i = j \text{ et } I_i[a_k] = I_j[a_k] \forall k \in [\![1, n]\!]. \end{cases}$$

Définition 3.27 Sous-flux partiel Un flux S_1 est dit *sous-flux partiel* de S_2 si le schéma de S_1 est une partie de S_2 et si S_1 est un sous-flux de S_2 lorsque l'on ne tient compte que de la partie du schéma commune aux deux flux. Formellement, soient $a_1, \cdots, a_n$ les noms d'attributs communs à S_1 et S_2 :

$$S_1 \subset_{a_1,\cdots,a_n} S_2 \Rightarrow \begin{cases} \mathcal{S}_1 \subset \mathcal{S}_2, \\ |S_1| < |S_2|, \\ \forall I_i \in \mathcal{I}_1, \exists I_j \in \mathcal{I}_2 \text{ tel que } I_i[a_k] = I_j[a_k] \forall k \in [\![1, n]\!]. \end{cases}$$

Toutes les règles de classification des sous-flux (continuité, ordre) s'appliquent aux sous-flux partiels : *sous-flux partiel ordonné*, noté $S_1 \overset{\leq}{\subset}_{a_1,\cdots,a_n} S_2$, et *sous-flux partiel continu*, noté $S_1 \sqsubset_{a_1,\cdots,a_n} S_2$.

3.1.5 Débit de flux

Définition 3.28 Débit d'un flux Le *débit d'un flux* caractérise le nombre d'éléments générés par le producteur du flux pour chaque période de temps. Le débit peut être *fixe* lorsque les éléments sont produits à une fréquence régulière ou *variable* lorsque la fréquence de production varie au cours du temps ; par exemple un flux d'évènements.

Le débit fixe d'un flux, noté ρ, est décrit par une simple valeur réelle tandis que le débit variable est décrit par une fonction. Soit $\mathcal{T}^+$ la fonction en escalier qui à tout temps t retourne le sous-flux continu d'éléments de S dont le temps de production est inférieur ou égal à t :

$$\mathcal{T}^+(t) = S^x \text{ tel que } \forall I \in S^x, \mathcal{T}(x) \leq t$$

Définition 3.29 Fonction de débit d'un flux La *fonction de débit* $\rho(t)$ décrit les variations du débit d'un flux S au cours du temps :

$$\rho(t) = \lim_{\Delta t \to 0} \frac{|\Theta^+(t)| - |\Theta^+(t - \Delta t)|}{t - \Delta t} \text{ et } |\Theta^+(t)| = \int_0^t \rho(t)dt$$

Le débit fixe d'un flux correspond à un débit variable pour lequel $\rho(t)$ est constante. Par généralisation, ρ désigne aussi bien un débit fixe que le débit moyen du flux sur toute sa durée :

$$\rho = \lim_{t \to +\infty} \frac{|\Theta^+(t)|}{t}$$

3.2 Modèle de calcul pour l'Internet des objets : traitement continu

La représentation des flux de données étant posée, nous nous intéressons à la représentation des applications orientées flux pour l'Internet des objets. Nous introduisons à cette fin une architecture orientée service adaptée au traitement continu et permettant de combiner aussi bien des services manipulant des ensembles finis de données que des services manipulant des flux de données.

3.2.1 Architecture orientée service pour le traitement continu

Les architectures orientées service (*service-oriented architecture*, ou *SOA*) ont été adoptées dans de nombreuses approches pour l'Internet des objets dans le but (i) d'abstraire les objets sous la forme de fournisseurs de services et (ii) de découpler les fonctionnalités de ces objets des technologies sous-jacentes [7]. Les principes qui sous-tendent les différentes *SOA* sont relatifs à la décomposition des applications monolithiques en composants plus petits et autonomes, ou *services*, dont les interfaces sont bien définies de façon à permettre leur réutilisation entre plusieurs applications. Typiquement, les *SOA* séparent les mécanismes permettant de démarrer l'exécution de ces composants (l'accès au service)

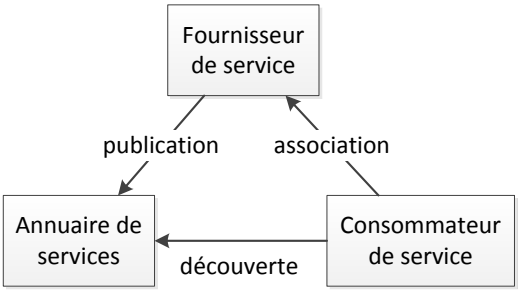


Figure 3.1 – Les rôles de base de l'architecture orientée service.

de la logique d'exécution proprement dite (l'implémentation du service). De fait, ces architectures sont particulièrement adaptées pour atténuer les problèmes d'hétérogénéité technique propres aux objets connectés.

De manière abstraite, une *SOA* se base sur trois rôles principaux, présentés sur la Figure 3.1 [45]. Dans ce cadre, un service est un ensemble d'opérations mises à disposition par un *fournisseur de service*, ou *hôte*. En parallèle, des *consommateurs de service*, ou *clients*, peuvent requérir l'exécution de ces opérations (invocation). En outre, les fournisseurs de services ont pour responsabilité de fournir une description, ou *contrat*, de chaque service disponible et de chaque opération associée. Cette description est publiée dans un *annuaire de services* permettant de mettre en relation fournisseurs et consommateurs (*découverte de services*). En effet, dans un scénario classique, un consommateur de service va (i) contacter l'annuaire, pour découvrir quels sont les fournisseurs de services proposant une fonctionnalité spécifique, puis (ii) s'associer [2] à l'un d'eux pour invoquer les opérations requises.

Typiquement, les *SOA* sont utilisées pour traiter des ensembles finis de données et ne sont pas directement exploitables pour traiter des flux continus de données [160]. Pour les besoins de notre approche de traitement continu, nous introduisons donc une nouvelle *SOA* permettant d'exploiter des services capables de traiter des ensembles discrets et des flux continus de données. À notre connaissance, cette problématique n'a été abordée dans la littérature que par le travail présenté dans [160]. Toutefois, cette approche ne questionne que le transport des flux, en l'occurence de manière asynchrone grâce à une architecture de type *publish-subscribe*. Cependant, au-delà de la diffusion des flux, une *SOA* nécessite de considérer la façon dont les services et leurs interfaces sont représentés, composés et orchestrés.

Définition 3.30 Service discret Un *service discret* est un composant logiciel qui possède un nom et qui fournit une ou plusieurs opérations de traitement pour des ensembles finis de données. L'ensemble des opérations disponibles est dit *interface du service discret*.

2. Ici, le terme « associer » (*bind*) est à rapprocher du terme « contrat » utilisé pour décrire un service, ce contrat spécifiant les interactions entre le fournisseur et le consommateur. En pratique, il s'agit simplement de préciser quels sont les paramètres que le consommateur doit fournir pour que le fournisseur puisse accomplir sa tâche.

Définition 3.31 Opération discrète Une *opération discrète* est une fonctionnalité fournie par un *service discret*. Une opération possède un nom, admet un ensemble de paramètres d'entrée et produit un ou plusieurs résultats en sortie. Les paramètres d'entrée et les résultats sont caractérisés par leurs noms et leurs domaines. Lorsqu'une opération discrète est invoquée, celle-ci traite les ensembles finis reçus en paramètre et produit de nouveaux ensembles finis en résultat.

Définition 3.32 Instance de service discret Une fois déployé sur un fournisseur de service, un service discret est associé à une adresse permettant d'invoquer ses opérations. Le couple *(service, adresse)* est appelé *instance du service discret*.

De la même façon qu'une *SOA* sépare l'implémentation d'un service de son interface [161], nous cherchons à découpler l'exécution virtuellement infinie d'un composant logiciel de traitement continu de l'interface permettant d'accéder aux données produites indéfiniment. Pour ce faire, nous introduisons les *services continus*, pour lesquels les paramètres d'entrée et les résultats ne sont pas des ensembles finis, mais des flux continus de données.

Définition 3.33 Service continu Un *service continu* est un composant logiciel qui possède un nom et qui consomme un ou plusieurs flux de données en entrée et produit un ou plusieurs flux en sortie. Pour ce faire, un service continu met à disposition zéro, un ou plusieurs *ports d'entrée* et zéro, un ou plusieurs de *ports de sortie*. L'ensemble des ports disponibles est dit *interface du service continu*.

Définition 3.34 Port d'entrée et port de sortie Un *port d'entrée* est utilisé pour connecter un ou plusieurs flux de mêmes schémas au service continu, ces différents flux étant traités en continu pour produire des résultats. Un *port de sortie* permet à un ou plusieurs consommateurs d'accéder à ces résultats au fur et à mesure de leur production, sous la forme de flux. Les propriétés des ports sont décrites dans la Section 3.2.1.1.

Définition 3.35 Port de sortie standard Tout service continu possède un *port de sortie standard*, ou *sortie standard*, ce port étant utilisé par défaut pour diffuser les résultats d'un traitement.

Définition 3.36 Instance de service continu Une fois déployé sur un fournisseur de service, un service continu est associé à une adresse permettant d'interagir avec ses ports. À la différence d'un service discret dont les opérations sont typiquement invoquées après instanciation, un service continu démarre ses traitements dès l'instanciation. De fait, lors de son instanciation, le service continu est informé des flux de données qui doivent être connectés à ses ports d'entrée. Le 4-uplet *(service, adresse, flux d'entrée, flux de sortie)* est appelé *instance du service continu*.

Propriété	Exemple
Nom de l'opérateur et éventuel paquetage.	*fr.inria.dioptase.stl.Union*
Noms des ports d'entrée.	*(port-principal)*
Noms des ports de sortie.	*(sortie-standard)*
Nombres minimum et maximum de flux autorisés par port.	$2 \leq$ *port-principal* $\leq +\infty$
Types sémantiques acceptés par port.	*(port-principal, tous)*
Types concrets acceptés par port.	*(port-principal, tous)*
Contraintes sur flux d'entrée.	$\varnothing$
Schéma de sortie.	*(sortie-standard, = port-principal)*
Paramètres d'instanciation.	$\varnothing$

Table 3.2 – Exemple de contrat.

3.2.1.1 Interface de service continu

L'interface d'un service continu est représentée par les ports mis à disposition par le service ; les ports d'entrée spécifiant les schémas des flux admissibles en entrée et les ports de sortie spécifiant les schémas des flux produits par le service continu.

Définition 3.37 Contrat de service continu Un *contrat de service continu* décrit l'ensemble des informations relatives au service continu et à son interface :

- le nom du service, le nom des ports d'entrée, le nom des ports de sortie ;
- le port de sortie à utiliser en tant que sortie standard ;
- le nombre minimum et maximum de flux autorisés pour chaque port ;
- les règles permettant de déterminer quels schémas de flux sont compatibles avec quels ports d'entrée ;
- les schémas des flux produits par les différents ports de sortie, ou les règles permettant d'inférer ces schémas à l'instanciation à partir des flux d'entrée ;
- le nom et le domaine des paramètres nécessaires à la configuration du service continu lors de son instanciation. Ces paramètres sont spécifiques à un service continu et à son implémentation.

À partir du contrat, il est possible (i) de raisonner à priori sur la nature du service continu et sur ses flux admissibles et (ii) de vérifier que toutes les contraintes sont satisfaites lorsque le service continu est déployé et instancié, c'est-à-dire configuré avec les paramètres fournis et connecté aux flux spécifiés. Un exemple de contrat est montré dans la Table 3.2, pour un service continu implémentant une union ensembliste.

Définition 3.38 Règles sur ports d'entrée Une règle sur les ports d'entrée est un prédicat qui, s'il est vérifié, indique que le port est compatible avec le schéma du flux. Deux types de règles sont considérées :

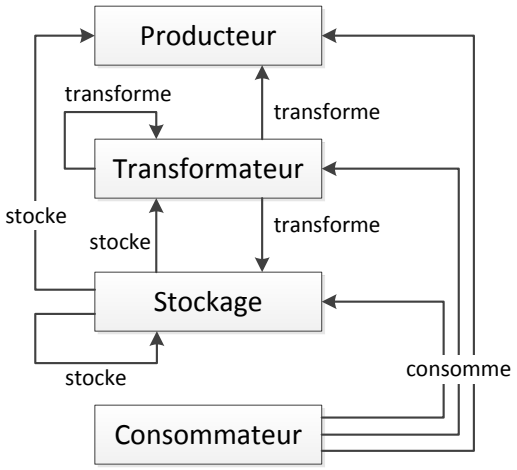


Figure 3.2 – Le modèle à quatre rôles.

— celles qui spécifient les schémas compatibles avec chaque port : *prédicat(nom de port, schéma de flux)* ;

— celles qui spécifient les contraintes globales sur les ports : *prédicat((nom du premier port, schéma de flux), ..., (nom du dernier port, schéma de flux))*.

Les règles de la première famille conditionnent les types sémantiques, les unités et les types concrets acceptés par le port tandis que les règles de la seconde famille permettent de spécifier des dépendances entre les ports ; par exemple, « le schéma du port X est égal au schéma du port Y » ou « l'attribut A du schéma du port X est du même type que l'attribut B du schéma du port Y ».

Définition 3.39 Règles sur ports de sortie Une règle sur un port de sortie est une fonction qui construit le schéma du flux de sortie. Cette règle est dite *statique*, si elle fixe le schéma du flux de sortie, et *dynamique* si le schéma de sortie dépend des schémas des flux d'entrée.

Deux règles dynamiques sont typiquement utilisées : (i) pour spécifier que les flux produits par un port de sortie possèdent le même schéma qu'un flux d'entrée reçu sur un port donné ou (ii) que le schéma des flux produits par un port de sortie est une combinaison d'attributs des schémas de plusieurs flux d'entrée reçus sur des ports donnés.

3.2.1.2 Familles de services continus : les quatre rôles

Au sein d'une application orientée flux pour l'Internet des objets, les objets physiques sont amenés à remplir des rôles bien spécifiques :

— un rôle de *production* consistant à présenter les données issues des différentes sources de données (p. ex., capteurs) aux autres objets, sous forme de flux ;

— un rôle de *transformation* consistant à acquérir des flux de données et à les transformer de façon à produire de nouveaux flux ;

— un rôle de *consommation*, consistant à acquérir des flux de données dont les informations permettent de piloter des périphériques (p. ex., actionneurs) ou d'alimenter des interfaces hommes-machines ;

— un rôle de *stockage*, consistant à acquérir des flux de données dans le but de les sauvegarder en mémoire ou sur des stockages persistants, afin de les présenter à nouveau sous forme de flux, à la demande.

Par extension, ces quatre rôles peuvent représenter l'ensemble des composants logiciels qui constituent une application orientée flux pour l'Internet des objets. Conformément à ces rôles, nous spécialisons le concept de service continu en quatre familles, présentées sur la Figure 3.2 :

Définition 3.40 Producteur Un *producteur* est un service continu qui assure le rôle de *production* et implémente la logique nécessaire pour exposer une source de données sous la forme de flux. Un producteur ne possède aucun port d'entrée mais possède un unique port de sortie : la sortie standard.

Définition 3.41 Transformateur Un *transformateur* est un service continu qui assure le rôle de *transformation* et implémente la logique nécessaire pour traiter et produire des flux. Un transformateur possède un ou plusieurs ports d'entrée et un ou plusieurs ports de sortie.

Définition 3.42 Stockage Un *stockage* est un service continu qui assure le rôle de *stockage* et implémente la logique nécessaire pour stocker, temporairement ou non, les éléments d'un ou de plusieurs flux. Un stockage possède un ou plusieurs ports d'entrée et un ou plusieurs ports de sortie utilisés pour récupérer les données stockées.

Définition 3.43 Consommateur Un *consommateur* est un service continu qui assure le rôle de *consommation* et implémente la logique nécessaire pour piloter des périphériques de sortie (p. ex. actionneurs) ou mettre à jour des interfaces hommes-machines à partir d'un ou de plusieurs flux. Un consommateur possède un ou plusieurs ports d'entrée mais ne possède aucun port de sortie.

Concrètement, les producteurs, les consommateurs et les stockages sont des interfaces vers d'autres systèmes ; respectivement (i) les sources de données (typiquement, des capteurs), (ii) les systèmes d'interactions avec l'environnement (actionneurs) ou l'utilisateur (interfaces hommes-machines) et (iii) les espaces de stockage (mémoire, fichier, base de données, etc.). Les transformateurs, quant à eux, expriment les logiques qui transforment des flux pour produire d'autres flux. Ces différentes familles de service continu permettent de représenter toutes les parties d'une application pour l'Internet des objets qui produit, traite, stocke et consomme des flux. Par exemple, les utilisateurs finaux, les interfaces graphiques qui affichent des résultats ou encore les actionneurs peuvent être représentés sous la forme de consommateurs. De la même façon, les capteurs, les bases de données, les *crowdsensors* et autres sources de données (par exemple, un service Web

	Ports d'entrée	Ports de sortie	Interface
Producteur	0	1	item **produce**()
Transformateur	1..*	1..*	**initialize**() item-list **work**(item) item-list **finalize**()
Stockage	1..*	1..*	**store**(item) item-list **produce**(query)
Consommateur	1..*	0	**consume**(item)

Table 3.3 – Interfaces logicielles minimales des différentes familles de service continu.

qui fournit des informations météorologiques) peuvent être abstraits par des producteurs. Pour finir, les transformateurs peuvent être utilisés pour implémenter n'importe quel type de traitement continu. Du fait de cette flexibilité, il est possible de représenter des applications complexes qui impliquent un grand nombre d'entités : capteurs, actionneurs, serveurs, utilisateurs, services tiers, etc.

Chaque famille de service continu nécessite, de la part des développeurs d'applications, d'implémenter une ou plusieurs interfaces logicielles minimales, listées dans la Table 3.3. La famille des producteurs possède une unique opération permettant de décrire la logique d'acquisition des données depuis une source de données et la logique d'encapsulation de ces données sous la forme d'éléments de flux. La famille des stockages possède (i) une opération permettant de décrire la logique de stockage des éléments de flux et (ii) une opération pour la diffusion des données stockées en réponse à une requête (voir le Chapitre 4 pour les requêtes supportées par l'implémentation actuelle). La famille des consommateurs possède une unique opération permettant de décrire comment les éléments de flux qui ont été consommés doivent être utilisés pour piloter des périphériques ou mettre à jour une interface homme-machine. Enfin, la famille des transformateurs possède un cycle de vie particulier lié à l'exécution des traitements continus sur les flux de données :

Définition 3.44 Cycle de vie d'un transformateur Le cycle de vie d'un transformateur est divisé en trois étapes : (i) le déploiement du transformateur, qui consiste à initialiser les ressources requises (variables globales, paramètres, etc.), (ii) le traitement de chaque nouvel élément reçu en entrée et (iii) l'arrêt du transformateur, qui consiste à effectuer des traitements finaux éventuels et à libérer les ressources précédemment initialisées.

Ces trois étapes sont représentées par les trois opérations constituant l'interface logicielle des transformateurs : (i) *logique d'initialisation*, (ii) *logique de traitement* d'un élément (ou *logique de travail*) et (iii) *logique de finalisation*. Chaque étape peut éventuellement lire et écrire des données au sein de l'état interne du transformateur, défini comme suit :

Définition 3.45 État interne d'un transformateur L'*état interne* d'un transformateur est une structure de donnée destinée à stocker les informations nécessaires à son exécution au cours du temps. Formellement, il s'agit d'un n-uplet Σ dont les attributs représentent les informations sauvegardées. La notation $\Sigma[k]$ peut désigner soit (i) la valeur du k-ème attribut du n-uplet Σ, si $k \in \mathbb{N}^*$, soit (ii) la valeur de l'attribut k du n-uplet Σ, si k est un nom d'attribut.

Définition 3.46 Fonction de construction de l'état interne Les attributs et la taille de l'état interne d'un transformateur sont potentiellement mis à jour à chaque nouvel élément lu depuis l'un des flux d'entrée S_1 à S_n. La *fonction de construction* σ décrit comment l'état Σ_i est transformé pour produire Σ_{i+1} lorsque de nouveaux éléments sont lus. Soient $X_1, \cdots, X_n$ les éléments en attente d'être traités, on a :

$$\Sigma_i = \begin{cases} \varnothing, \text{ si } i = 0, \\ \sigma(\Sigma_{i-1}, X_1, \cdots, X_n), \text{ sinon.} \end{cases}$$

Il est important de caractériser la croissance d'un état interne, dans le but de déterminer quelle quantité d'espace sera nécessaire pour le stocker ou le transférer :

Définition 3.47 Fonction de croissance d'un état interne La *fonction de croissance* σ^+ d'un état interne Σ décrit l'évolution de la taille de l'état interne au cours du temps : $\sigma^+(x) = |\Sigma_x|$.

Le comportement de σ^+ est analysable en utilisant la notion classique de domination asymptotique et les classes de croissances usuelles en notation de Landau : $O(1)$, $O(\log n)$, $O(n \log n)$, $O(n)$, $O(n^k)$, $O(k^n)$, $O(n!)$, etc.

3.2.1.3 Composition de services continus

Les *SOA* permettent de combiner les opérations de services existants pour former des *compositions de services* [45]. En effet, les entrées et les sorties des opérations étant spécifiées par les contrats de service, il est possible de créer un processus décrivant l'enchaînement des invocations des différentes opérations, ce processus formant la logique d'une nouvelle opération plus large. Ce mécanisme de composition est particulièrement intéressant du point de vue de l'Internet des objets, notamment pour combiner les fonctionnalités de plusieurs objets dans le but d'en créer dynamiquement de nouvelles qui ne seraient pas disponibles nativement [46]. De la même façon, les services continus peuvent être combinés pour former des tâches et des applications :

Définition 3.48 Tâche de traitement continu Une *tâche de traitement continu*, ou tâche, est un ensemble de traitements que l'on souhaite appliquer à des données. Une tâche se compose d'un ensemble de services continus connectés entre eux ; les données à traiter sont générées par les producteurs et circulent au travers des transformateurs et des stockages jusqu'aux consommateurs.

Un *graphe* est un ensemble de *sommets* (*vertex*) reliés par des *arêtes*. Dans sa forme générale, il est noté $G = (V, E, \varphi)$ où V est l'ensemble des sommets, E l'ensemble des arêtes et $\varphi : E \to V \times V$ la *fonction d'incidence* qui à chaque arête associe un sommet de départ et un sommet d'arrivée, $\varphi(e) = \{x, y\}$, autorisant ainsi les arêtes parallèles entre deux sommets (*multigraphe*) ou les *boucles* (arêtes dont le départ et l'arrivée sont les mêmes). On appelle *sources* les sommets du graphe ne possédant aucun *prédécesseur* et *puits* les sommets ne possédant aucun *successeur*. On appelle *ordre* du graphe le cardinal de V, noté $|V|$ et $|E|$. On appelle *degré d'un sommet* x, noté $d(x)$, le nombre d'arêtes dans lesquelles x est impliqué et *degré maximal* (resp. minimal) du graphe le maximum (resp. minimum) de $d(x)$ $\forall x \in V$.

Un graphe *simple*, c'est-à-dire sans arêtes parallèles ni boucles, peut être noté $G = (V, E)$ où E est alors directement considéré un ensemble de paires non ordonnées $\{x, y\} \in V \times V$. Dans ce cadre, on appelle *matrice d'adjacence* la matrice carrée A de dimension $V \times V$ où $A_{ij} = 1$ si $\{v_i, v_j\} \in E$, 0 sinon.
Un graphe est dit *orienté* (ou *dirigé*) si l'on donne une direction aux arêtes (alors appelée *arc*), c'est-à-dire lorsque l'on distingue une arête (x, y) d'une arête (y, x).
Un graphe est dit *acyclique* s'il n'existe aucun *cycle*, c'est-à-dire aucune suite d'arêtes consécutives dont les sommets de départ et de fin sont les mêmes.

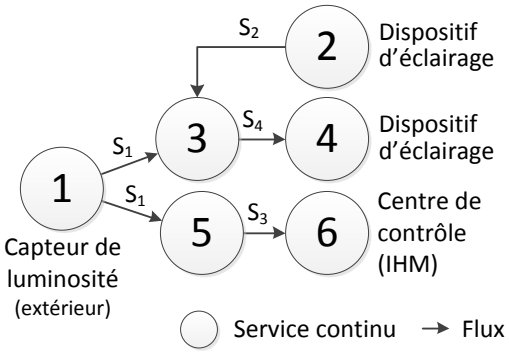
Encart 3.3 – Rappel sur les notations de graphe.

Définition 3.49 Application orientée flux Une *application orientée flux* est un logiciel applicatif comportant au moins une tâche de traitement continu.

Concrètement, le mécanisme de composition de services continus se base sur la description du flot des données entre les services continus qui constituent une tâche, sous la forme d'un graphe logique :

Définition 3.50 Graphe logique Un *graphe logique* est un graphe acyclique dirigé $GL = (VL, EL)$, où $VL = \{vl_i\}_{1 \leq i \leq |VL|}$ est un ensemble de services continus et $EL \subseteq VL^2$ un ensemble d'arcs qui relient deux services continus entre eux. Chaque arc représente un flux, noté S_{ij}, qui circule d'une opération vl_i vers une opération vl_j : $(vl_i, vl_j) \in EL$.

Par définition, un graphe logique décrit une tâche qui se compose de producteurs (sources de GL), de transformateurs, de stockages et de consommateurs (puits de GL). La Figure 3.3 présente, pour exemple, un graphe logique décrivant une tâche consistant à analyser la lumière extérieure pour piloter un éclairage intelligent au sein d'une pièce. Dans cette tâche, un producteur mesure la luminosité extérieure tandis qu'un autre producteur analyse les changements d'état du dispositif d'éclairage (évènements *on* ou *off*). Ces informations sont exploitées par un transformateur qui produit alors un flux d'ordres *switch-on* ou *switch-off* à destination du dispositif d'éclairage (consommateur). En parallèle, les différentes mesures de luminosité sont enregistrées dans un stockage et



Famille	Description
① Producteur	Lit le capteur de lumière toutes les x secondes et produit une valeur de luminosité.
② Producteur	Produit l'évènement *on* lorsque le dispositif d'éclairage s'allume, ou *off* lorsque le dispositif d'éclairage s'éteint.
③ Transformateur	Produit l'évènement *switch-off* si ② produit *on* et si la valeur de luminosité est supérieure à un seuil.
	Produit l'évènement *switch-on* si ② produit *off* et si la valeur de luminosité est inférieure ou égale à un seuil.
④ Consommateur	Allume (resp. éteint) l'éclairage si *switch-on* (resp. *switch-off*) est reçu depuis ③.
⑤ Stockage	Enregistre chaque valeur de luminosité (p. ex. dans un fichier).
⑥ Consommateur	Contacte ④ périodiquement pour récupérer les données stockées et les affiche dans l'application de contrôle.

Figure 3.3 – Un graphe logique représentant une tâche de contrôle d'éclairage.

constituent un historique qui alimente un autre consommateur : l'application de gestion du dispositif d'éclairage dont le but est de présenter ces données historiques à l'utilisateur.

Définition 3.51 Exécution d'un graphe logique L'exécution d'un graphe logique GL consiste à (i) déployer les différents services continus qui le composent et (ii) à connecter leurs ports entre eux conformément aux arcs de GL. Ce déploiement peut être réalisé soit par l'application elle-même, soit au travers d'un intermédiaire : un *serveur de déploiement*.

Le Chapitre 4 porte spécifiquement sur les détails relatifs à l'exécution des graphes logiques.

3.2.2 Langage de traitement de flux

En pratique, l'implémentation d'un transformateur sur une plateforme donnée peut se faire de deux façons :

— soit en utilisant les langages natifs fournis par cette plateforme, de la même façon que pour les systèmes génériques de traitement de flux ;

— soit au travers d'un langage dédié (*DSL*), compilé à destination d'une plateforme donnée ou directement interprété par cette plateforme.

Les expressions symboliques [162], ou *S-Expressions*, désignent une notation pour la description de structures arborescentes, notamment utilisée pour décrire aussi bien des données que des logiques d'exécutions dans le langage fonctionnel *LISP* et ses dérivés. Concrètement, une *S-Expression* peut représenter soit un littéral, ou *atome*, soit une composition de *S-Expressions* délimitées par des parenthèses ; par exemple, une expression arithmétique telle que *3 + 2 - 5* pourra être représentée par les *S-Expressions (3 + 2 - 5)* ou *(- (+ 3 2) 5)*, la seconde notation, ou *notation préfixée*, étant celle employée par *LISP*.

Encart 3.4 – Rappel sur les *S-Expressions*.

Comme nous l'avons vu au Chapitre 2, la littérature sur les *DSMS* introduit déjà des langages de traitement de flux. Toutefois, nous cherchons à exploiter les capacités de tous les objets disponibles et, pour ce faire, souhaitons que les objets soient en mesure d'héberger leurs propres services continus (*in-network processing*) sans nécessiter une infrastructure d'exécution dédiée comme cela pourrait être le cas dans une approche *cloud of Things*. Par ailleurs, cette capacité à exécuter localement des services continus doit aussi être assurée lorsque ceux-ci sont exécutés par un interpréteur visant à accroître leur dynamicité et leur portabilité.

Comme les objets visés par notre approche sont potentiellement limités en ressources, nous introduisons un nouveau langage dédié au traitement de flux, *DiSPL* (*Dioptase Stream Processing Language*). Les objectifs sous-jacents de ce langage sont les suivants :

1. réduire au minimum le coût de l'analyse syntaxique d'un programme et sa représentation en mémoire ;
2. intégrer les concepts propres à nos modèles de données et de calcul présentés précédemment.

En rapport avec l'objectif (1), *DiSPL* est basé sur la sémantique et la grammaire du langage *Scheme* [163]. En effet, l'interprétation de *Scheme* peut être réalisée efficacement en matière d'occupation mémoire et d'analyse syntaxique, et ce pour deux raisons :

— *Scheme* utilise des expressions symboliques (*S-Expressions*) [162] (voir Encart 3.4) pour représenter les programmes ; l'avantage principal de ces expressions réside dans leur grammaire réduite pour laquelle il est possible de produire un analyseur syntaxique très léger.

— *Scheme* a été voulu très épuré par ses créateurs et possède peu de mots-clés [163].

DiSPL est un langage généraliste permettant de décrire toutes sortes d'opérations complexes en utilisant des constructions classiques du langage *Scheme* (conditions, opérations arithmétiques et logiques, etc.), auquel *DiSPL* ajoute les boucles *for* et *while* afin d'en faciliter l'adoption. Cependant, à la différence de *Scheme*, *DiSPL* est aussi un *DSL* pour le traitement de flux et intègre, à cette fin, des fonctions primitives relatives à notre modèle de données et de calcul. Outre ces fonctionnalités, nous n'introduisons pas de nouveautés particulières du point de vue des langages de traitement de flux existants,

```
1  ; entête du programme
2  ... code DiSPL pour charger les bibliothèques ...
3  ... code DiSPL pour déclarer le contrat du transformateur ...
4
5  init:   ; section d'initialisation
6  ... code DiSPL ...
7  work:   ; section de travail
8  ... code DiSPL ...
9  end:    ; section de finalisation
10 ... code DiSPL ...
```

Figure 3.4 – Les quatre parties d'un programme *DiSPL*.

l'objectif principal de *DiSPL* étant de faciliter la description de programmes pouvant être analysé et interprété efficacement sur des appareils limités en ressources.

Contrairement à un programme *Scheme*, un programme *DiSPL* est découpé en quatre parties, comme présenté sur la Figure 3.4 : un entête suivi de trois *sections* qui correspondent chacune à une étape du cycle de vie d'un transformateur.

Définition 3.52 Entête d'un programme *DiSPL* L'entête d'un programme *DiSPL* correspond à l'ensemble des instructions qui se situent au début du programme et qui indiquent (i) quels sont les paquetages utilisés par le programme (instruction *include*) et (ii) quel est le contrat du programme (instruction *contract*, présentée dans la Section 3.2.2.1).

Définition 3.53 Section d'initialisation La section d'initialisation décrit les instructions qui doivent être exécutées dès que le transformateur est instancié. Les variables déclarées dans cette section sont globales au programme *DiSPL* et constituent l'état interne du transformateur. Les instructions définies dans les autres sections peuvent accéder directement à ces variables par leur nom.

Définition 3.54 Section de travail La section de travail décrit les instructions à exécuter à chaque fois que le transformateur est prêt à traiter un élément reçu en entrée. Dans cette section, le développeur peut accéder aux éléments qui sont en attente de traitement et écrire sur les différents flux de sortie, au moyen des primitives de gestion de flux (voir Section 3.2.2.2). Contrairement à la section d'initialisation, les variables déclarées dans cette section sont locales.

Définition 3.55 Section de finalisation La section de finalisation décrit les instructions qui doivent être exécutées avant de détruire le transformateur. Il s'agit typiquement de libérer des ressources ou de finaliser des traitements. La section de finalisation peut accéder aussi bien aux éléments en attente qu'aux flux de sortie. Les variables déclarées dans cette section sont locales.

De ces trois sections, seule la section de travail est obligatoire dans un programme *DiSPL*.

3.2.2.1 Expression et manipulation des types primitifs

Tout comme *Scheme*, *DiSPL* gère les types des variables à l'exécution et non pas à l'écriture du programme. Cependant, *DiSPL* permet de ne manipuler que les types primitifs que nous avons définis pour les types concrets : entier, réel, booléen, chaîne de caractère et tableau d'octets (*blob*). *DiSPL* gère les collections de la même façon que *Scheme*, notamment la distinction entre une liste (liste chaînée) et un vecteur (tableau unidimensionnel fixe) [163].

DiSPL introduit l'ensemble des types propre au modèle de données et au modèle de calcul que nous avons présentés précédemment : flux, schéma de flux, élément de flux et contrat de service continu. Déclarer et manipuler ces types se fait au travers d'un ensemble de macro-instructions qui servent de façade au mécanisme de liste d'association (*association list*) de *Scheme* [163].

Le contrat d'un transformateur est toujours déclaré dans l'entête d'un programme *DiSPL*. Ce contrat comporte quatre parties : les paramètres d'instanciation, les ports d'entrée, les ports de sortie et les métadonnées définies par le développeur.

```
(contract (package <nom de package>) (name <nom du transformateur>)
    (parameters
        (parameter (name <nom du paramètre>)
            (type <type du paramètre>)
            (domain <expression de domaine>)
        )
        <autres instructions parameter>
    )
    (input-ports
        (port (name <nom du port>)
            (min <nombre minimum de flux ou "infinity">)
            (max <nombre maximum de flux ou "infinity">)
            (accept-rules
                (input-rule <règle sur port d'entrée>)
                <autres instructions input-rule>
            )
        )
        <autres instructions port>
    )
    (output-ports
        (port (name <nom du port>)
            (standard-output <1 ou 0>)
            (output-rule <règle sur port de sortie>)
        )
        <autres instructions port>
    )
    (metadata <métadonnées définies par le développeur>)
)
```

Une fois ce contrat déclaré, le reste du programme peut faire référence aux paramètres et aux métadonnées par leur nom grâce aux macro-instructions *metadata-name* et *parameter-name*. Par exemple, *(parameter-name "primary-key")* sera remplacé à l'instanciation par la valeur du paramètre dont le nom est *primary-key*.

Trois contraintes de domaine sont exprimables : un intervalle de valeurs, un ensemble de valeurs autorisées (*one-of*) et un ensemble de valeurs interdites (*none-of*).

```
(domain
    (min <valeur minimum, "-infinity">)
    (max <valeur maximum ou "+infinity">)
)
(domain
    (one-of <liste de valeurs autorisées>)
)
(domain
    (none-of <liste de valeurs interdites>)
)
```

Par ailleurs, une contrainte de domaine peut être composite, c'est-à-dire agréger plusieurs contraintes de domaines. À noter que, en cas de conflit, les contraintes d'exclusion de valeur (*none-of*) sont toujours prioritaires par rapport aux contraintes d'inclusion de valeur (*one-of* et intervalle).

```
(domain
    (domain <expression de domaine>)
    <autres instructions domain>
)
```

Conformément à ce qui a été défini à la Section 3.2.1.1, trois types de règles sont exprimables sur un port d'entrée :

— *Règle sur le schéma des flux admis par le port*, qui définit quels sont les attributs requis. Pour chaque attribut requis, un ensemble de types sémantiques, de types concrets et d'unités peuvent autorisés (*one-of*) ou interdits (*none-of*) pour le port. Les différentes valeurs de type et d'unité sont des chaînes de caractères, dans lesquelles le métacaractère « * » peut être utilisé pour se substituer à zéro, un ou plusieurs caractères ; par exemple « image/* ».

```
(input-rule (type "schema")
    (attribute
        (semantic-type <liste de types sémantiques>)
        (unit <liste d'unités autorisées>,)
        (concrete-type <liste de types concrets>)
    )
    <autres instructions attribute>
)
```

— *Règle d'égalité de schéma*, qui définit quels sont les ports qui devront posséder le même schéma :

```
(input-rule (type "same-schema")
    (ports <liste de noms de ports>)
)
```

— *Règle d'égalité d'attribut*, qui définit quels sont les ports qui devront posséder un même attribut :

```
(input-rule (type "same-attribute")
    (attribute <nom ou position d'attribut>)
    (ports <liste de noms de ports>)
)
```

Enfin, trois types de règles sont exprimables sur un port de sortie, tel que formalisé à la Section 3.2.1.1 :

— *Règle de schéma statique*, qui définit explicitement le schéma des flux de sortie produits par le port :

```
(output-rule (type "schema")
    <déclaration de schéma, voir instruction schema>
)
```

— *Règle d'identité de schéma*, qui définit que le schéma des flux de sortie est le même que le schéma d'un des ports d'entrée :

```
(output-rule (type "identity")
    (port <nom du port d'entrée>)
)
```

— *Règle de composition de schéma*, qui définit comment plusieurs schémas de port d'entrée peuvent être composés entre eux. La règle de composition de schéma construit un espace d'attributs à partir (i) des attributs d'un ou de plusieurs schémas d'entrée et (ii) de nouveaux attributs déclarés explicitement. Les deux macro-instructions *included-attribute* et *excluded-attribute* permettent d'inclure ou d'exclure spécifiquement certains attributs de cet espace, à partir de leurs noms.

```
(output-rule (type "composition")
    (port <nom du port>)
    (attribute <déclaration d'attribut, voir instruction schema>)
    (excluded-attribute <nom de port> <nom d'attribut>)
    (included-attribute <nom de port> <nom d'attribut>)
)
```

Outre la description du contrat, *DiSPL* permet de construire et de manipuler des flux, des schémas et des éléments de flux :

```
(stream (uri <URI du flux>)
    (schema <déclaration du schéma>)
)

(schema
    (attributes
        (attribute (name <nom de l'attribut>)
            (semantic-type <type concret>)
            (unit <unité>)
            (concrete-type <type concret>)
            (domain <expression de domaine>)
            (metadata <métadonnées définies par le développeur>)
        )
        <autres instructions attribute>
    )
    (metadata <métadonnées définies par le développeur>)
)

(item (timestamp <timestamp de l'élément>)
    (<nom du premier attribut> <valeur du premier attribut>)
    <autres attributs>
    (<nom du dernier attribut> <valeur du dernier attribut>)
)
```

Fonction	Rôle
(get-new-items <URI du flux>)	Récupère l'ensemble des éléments présent dans la file d'attente pour un flux donné.
(get-next-item <URI du flux>)	Récupère l'élément le plus récent de la file d'attente pour un flux donné.
(get-next-items <URI du flux> <nombre d'éléments à lire>)	Récupère les n éléments les plus récents de la file d'attente pour un flux donné.
(write <nom du port> <élément de flux>)	Écrit un élément dans tous les flux connectés au port de sortie spécifié.

Remarque : Le nom du port d'entrée peut être utilisé à la place des *URI* de flux, auquel cas le premier flux connecté au port sera lu.

Table 3.4 – Fonctions primitives de lecture et d'écriture de flux.

3.2.2.2 Fonctions primitives de gestion de flux et d'instance

Dans les sections de travail et de finalisation d'un programme *DiSPL*, le programme est autorisé à lire les flux d'entrée et à écrire sur les flux de sortie. Chaque flux d'entrée est associé à un *URI* composée du nom du service continu et du nom du port à l'origine du flux. Cet *URI* est utilisé par les fonctions primitives de lecture des flux d'entrée pour accéder aux éléments en attente de traitement. Lorsqu'une de ces primitives extrait un élément depuis la file d'attente, celui-ci est marqué comme lu et sera ignoré si les fonctions de lecture de flux sont appelées à nouveau. Lorsque la section de travail se termine, tous les éléments marqués lus sont alors supprimés de la file d'attente. Les primitives de lecture de flux sont décrites dans la Table 3.4.

Dans chacune de ses sections, un programme *DiSPL* peut accéder aux informations paramétrées à l'instanciation du transformateur : (i) les valeurs des paramètres, (ii) le schéma réel des flux qui ont été connectés aux ports d'entrée et (iii) le schéma réel des ports de sortie. L'instance du transformateur est décrite par la structure suivante :

```
(processor-instance
    (contract <le contrat tel que défini dans l'entête>)
    (parameters
        (<nom du premier paramètre> <valeur du premier paramètre>)
        <autres paramètres>
        (<nom du dernier paramètre> <valeur du dernier paramètre>)
    )
    (input-ports
        (<nom du premier port d'entrée> (streams <liste des flux associés>))
        <autres ports d'entrée>
        (<nom du dernier port d'entrée> (streams <liste des flux associés>))
    )
    (output-ports
        (<nom du premier port de sortie> (schema <schéma associé>))
        <autres ports de sortie>
        (<nom du dernier port de de sortie> (schema <schéma associé>))
    )
)
```

Fonction	Rôle
(stdout)	Retourne le nom du port de sortie standard.
(this)	Retourne les informations sur l'instance du transformateur.
(get-stream <URI de flux>)	Retourne les informations sur un flux à partir de son *URI*.

Table 3.5 – Fonctions primitives d'accès aux informations de l'instance du service continu.

Les primitives d'accès aux informations de l'instance sont décrites dans la Table 3.5.

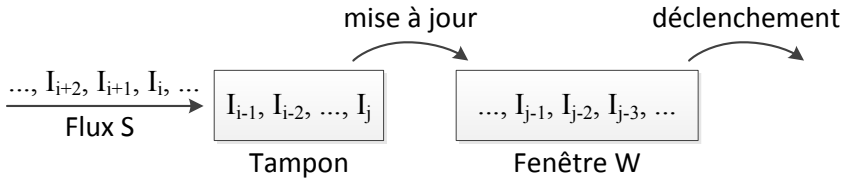
3.2.2.3 Exemple de programme *DiSPL*

L'exemple de programme *DiSPL* qui suit décrit l'implémentation d'une simple fonction de comptage continu en tant que transformateur. Étant donné un ou plusieurs nouveaux éléments reçus en entrée, le programme met à jour un compteur et écrit la valeur de ce compteur dans les flux de sortie.

L'entête du programme *DiSPL* déclare le contrat du transformateur, qui possède un seul port d'entrée admettant un seul flux de structure quelconque ; pas de règle sur le port d'entrée. Le transformateur n'admet aucun paramètre et possède un unique port de sortie, nommé *count-result*. Le schéma associé aux flux produits par ce port est fixé statiquement et décrit des éléments possédant la valeur du compteur comme seul attribut. Cette valeur n'ayant ni sémantique ni unité particulière, seul le type concret est spécifié, correspondant au type entier (*primitive/integer*). Un seul port étant défini, il n'est pas non plus nécessaire de préciser explicitement qu'il s'agit du port de sortie standard.

```
(contract (name "count-processor")
    (input-ports
        (port (name "main-port")                ; un seul port d'entrée, avec un seul
            (min 1) (max 1)                     ; flux autorisé.
        )
    )
    (output-ports
        (port (name "count-result")             ; un seul port de sortie avec un seul
            (output-rule (type "schema")        ; attribut de type entier, sans type
                (schema                         ; sémantique.
                    (attribute (name "count")
                        (concrete-type "primitive/integer")
                    )
                )
            )
        )
    )
)
```

L'entête se termine implicitement dès la rencontre de la section d'initialisation. Le code *DiSPL* de cette section initialise un compteur global qui constitue l'état interne du transformateur. Dans l'extrait de code qui suit, la section de travail décrit le traitement des éléments reçus sur l'unique flux du port d'entrée : (i) lecture des éléments en attente, (ii) comptage et (iii) écriture d'un nouvel élément de flux sur le port de sortie standard.



Les nouveaux éléments sont stockés dans un tampon jusqu'à la prochaine mise à jour de la fenêtre (ajout-suppression d'éléments), puis, après plusieurs mises à jour, le contenu est retourné par la fenêtre et peut être traité.

Figure 3.5 – Fonctionnement d'une fenêtre.

```
init:                                          ; section d'initialisation.
    (let count 0)                              ; création d'un compteur global.
work:                                          ; section de travail.
    (let diff (get-new-items "main-port"))     ; récupère les nouveaux éléments.
    (cond ((> (length diff) 0)
        (set! count (+ count (length diff)))   ; calcule un nouveau total.
        (write (stdout)                        ; écrit le total sur la sortie standard.
            (item
                (timestamp (now))
                ("count" count)
            )
        ))
    )
```

Un exemple de programme *DiSPL* plus complexe est présenté dans l'Annexe 1, consistant en une implémentation d'un *filtre de Bloom* [164].

3.2.3 Opérateur de fenêtre

Nous avons vu, dans la Section 2.2.1.2 du Chapitre 2, qu'une fenêtre est un opérateur de traitement de flux qui construit un ensemble fini d'éléments à partir d'un flux. Les fenêtres sont des opérateurs indispensables à l'exécution de traitement bloquant ; par exemple, les tris de données.

Définition 3.56 État d'une fenêtre À un instant donné, une fenêtre est caractérisée par son *état*, ou *contenu*, c'est-à-dire la suite d'éléments extraits du flux S. L'état d'une fenêtre forme un sous-flux de S.

Définition 3.57 Mise à jour d'une fenêtre La *mise à jour d'une fenêtre*, ou *changement d'état*, est l'étape consistant à actualiser le contenu de la fenêtre : ajout d'éventuels nouveaux éléments et retrait des éléments qui ne satisfont plus les paramètres de la fenêtre (voir Figure 3.5). Un changement d'état peut survenir au fur et à mesure de la lecture du flux ou à intervalles réguliers.

Définition 3.58 Déclenchement d'une fenêtre Le *déclenchement d'une fenêtre* est l'évènement qui survient lorsqu'une fenêtre se trouve dans un certain état, pour lequel on considère que le contenu est prêt à être traité. Lorsqu'une fenêtre est déclenchée,

Mise à jour	**Prédicat** α	
périodique positionnelle	$\alpha(S^{a\to b})$	$\lvert(S^{a\to b})\rvert \geq k$, avec k constant.
périodique temporelle*	$\alpha(t)$	t **mod** $k = 0$, avec k constant et t le temps courant.
à chaque nouvel élément	$\alpha(S^{a\to b})$	$\lvert S^{a\to b}\rvert > 0$.
à chaque variation de temps (flux temporels uniquement)	$\alpha(S^{a\to b})$	$\mathcal{T}(I_b) \neq \mathcal{T}(I_{b-1})$.
à chaque variation de la valeur d'un attribut	$\alpha(S^{a\to b})$	$I_b[k] \neq I_{b-1}[k]$ avec k le k-ème attribut de chaque élément.

* mod = modulo.

Table 3.6 – Différents modes de mise à jour α.

l'ensemble fini d'éléments qu'elle a construit est rendu accessible aux autres traitements (voir Figure 3.5).

3.2.3.1 Formulation générale des fenêtres

Le comportement d'une fenêtre se représente au travers de quatre fonctions modélisant les différentes phases de son évolution : un prédicat de mise à jour α, une fonction d'évaluation β, une fonction de transformation γ et un prédicat de déclenchement δ :

Définition 3.59 Prédicat de mise à jour α Soit $S^{a\to b}$ la portion du flux S contenant les éléments lus mais non traités par la fenêtre. Le *prédicat de mise à jour* α permet de déterminer à quel moment le contenu de la fenêtre doit être mis à jour avec les éléments de $S^{a\to b}$. Les prédicats de mise à jour usuels sont présentés dans la Table 3.6.

Définition 3.60 Fonction d'évaluation β Étant donné un élément I, la *fonction d'évaluation* $\beta : \mathcal{I} \longrightarrow \{0, 1\}$ indique si I doit être conservé pour la mise à jour de la fenêtre.

Définition 3.61 Fonction de transformation γ Soit w_i l'état courant de la fenêtre et $S^{a\to b}$ la portion du flux S contenant les éléments lus mais non traités. La *fonction d'évaluation* γ construit un nouvel état w_{i+1} à partir des éléments contenus dans w_i et dans $S^{a\to b}$.

Définition 3.62 Prédicat de déclenchement δ Le *prédicat de déclenchement* δ permet de déterminer à quel moment le contenu w_i de la fenêtre est prêt à être produit pour traitement ultérieur. Les différents prédicats de déclenchement usuels sont présentés dans la Table 3.7.

Étant donné β et γ, on peut représenter une fenêtre comme une succession d'état w_i, où w_i est calculé à partir de l'état précédent et du tampon $S^{a\to b}$. Soit $X = (w \parallel S^{a\to b})$ la concaténation du contenu w et du tampon $S^{a\to b}$, on a :

$$w_i = \begin{cases} \varnothing, \text{ si } i = 0, \\ \gamma(X_{i-1}), \text{ c'est-à-dire la suite } (I_j) \text{ constituée des } I_j \in X_{i-1} \text{ tels que } \beta(I_j) = 1. \end{cases}$$

Transmission	**Prédicat** δ	
au changement d'état	$\delta(w_{i-1}, w_i)$	$w_{i-1} \neq w_i$.
à la fermeture	$\delta(p_s, p_e)$	$p_s \geq p_e$.
à la fenêtre pleine	$\delta(w_i)$	$\lvert w_i \rvert \geq \lvert W \rvert_{max}$.
à la fenêtre non vide	$\delta(w_i)$	$\lvert w_i \rvert > 0$.
périodique positionnel*	$\delta(w_{i-1}, w_i)$	abs($\lvert w_{i-1} \rvert - \lvert w_i \rvert$) > 0.
périodique temporel*	$\delta(t)$	t **mod** $k = 0$, avec k constant et t le temps courant.

* mod = modulo, abs = valeur absolue.

Table 3.7 – Différents prédicats de déclenchement δ.

Spécifiquement, la fonction d'évaluation β définit les bornes d'une fenêtre et, par extension, sa taille :

Définition 3.63 Bornes inférieure et supérieure de la fenêtre La *borne inférieure* p_s, ou *point de départ*, et la *borne inférieure* p_e, ou *point d'arrivée*, sont des valeurs sélectionnées dans un ordre supplémentaire $\mathcal{R}$ du flux, telles que $p_s < p_e$. Les bornes p_s et p_e peuvent prendre trois formes :

- *Borne fixe* : une valeur $p \in \mathcal{R}$.
- *Borne mobile* : le présent de l'ordre $\mathcal{R}$, noté $\dot{p}$.
- *Borne mobile relative* : une expression de la forme $\dot{p} + k$ où k est une constante ; par exemple $\dot{p} - 10$mn.

Définition 3.64 Taille logique d'une fenêtre La *taille logique d'une fenêtre* correspond à sa taille exprimée par rapport à l'ordre $\mathcal{R}$ (p. ex. 10 minutes) : elle est égale à $p_e - p_s$.

Définition 3.65 Taille physique d'une fenêtre La *taille physique d'une fenêtre*, notée $|W|$, correspond au nombre d'éléments qu'elle contient, indépendamment de l'ordre $\mathcal{R}$.

Dans certains cas, la taille logique et la limite sont confondues, par exemple si $\mathcal{R}$ est l'ensemble des positions. En effet, il serait absurde de définir une fenêtre d'une taille logique de x éléments et de la limiter physiquement à un nombre d'éléments inférieur ou supérieur à x.

Définition 3.66 Fenêtre pleine On dira qu'une fenêtre est *pleine* si sa taille physique atteint une limite $|W|_{max}$ donnée.

3.2.3.2 Fenêtre positionnelle et fenêtre temporelle

Étant donné que nous avons formalisé les fenêtres sous une forme générale, nous préférerons parler de *fenêtre positionnelle* plutôt que de fenêtre physique lorsque $\mathcal{R}$ est l'ensemble des positions. De la même façon nous préférerons le terme *fenêtre temporelle* à celui de fenêtre logique, lorsque $\mathcal{R}$ est l'ensemble temps.

Définition 3.67 Fenêtre positionnelle Une *fenêtre positionnelle* W_φ correspond à une fenêtre basée sur l'ordre positionnel :

$$W_\varphi = \begin{cases} \mathcal{R} = \mathbb{N}^*, \\ \dot{p} \text{ est la position du dernier élément ajouté}, \\ |W|_{max} = p_e - p_s, \\ \beta(I_i) = 1 \text{ si } p_s \leq i \leq p_e, 0 \text{ sinon}. \end{cases}$$

La fenêtre positionnelle est le type de fenêtre le plus simple et le plus général, car se basant sur l'ordre positionnel. En effet, cet ordre étant défini naturellement par la position des éléments du flux dans la suite $\mathcal{I}$, il est toujours respecté même dans un flux quelconque. Par ailleurs, l'ordre positionnel est assimilable à tous les ordres supplémentaires du flux. Enfin, la limite physique de ces fenêtres est implicite, étant égale à sa taille logique et à sa taille physique une fois pleine.

Définition 3.68 Fenêtre temporelle Une *fenêtre temporelle* W_θ correspond à une fenêtre basée sur l'ordre temporel :

$$W_\theta = \begin{cases} \mathcal{R} = \mathbb{T}, \\ \dot{p} \text{ est le temps courant}, \\ |W|_{max} = k \text{ constant, ou } |W|_{max} = +\infty, \\ \beta(I_i) = 1 \text{ si } p_s \leq \mathcal{T}(I_i) \leq p_e, 0 \text{ sinon}. \end{cases}$$

La fenêtre temporelle W_θ est plus spécifique, ne s'appliquant qu'aux flux temporels. Cette fois-ci, la limite physique de la fenêtre doit être spécifiée explicitement. Elle peut cependant être infinie, ce qui implique une taille physique variable à chaque mise à jour, conformément au nombre d'éléments acceptés par la fonction d'évaluation β.

On constate qu'il existe une ambiguïté dans la sélection des éléments pouvant faire partie d'une mise à jour lorsque la limite physique est constante et que l'ensemble temps n'est pas un ordre supplémentaire total sur le flux, c'est-à-dire lorsque le flux n'est pas t-identifié et que plusieurs éléments ont un même timestamp. En effet, si la fenêtre est physiquement limitée à k éléments et que $k+x$ éléments ($x > 0$) de même timestamp sont stockés dans le tampon, il est nécessaire de déterminer quels éléments seront sélectionnés lors de cette mise à jour. Par défaut, puisque l'ensemble temps est un ordre supplémentaire du flux, l'ordre positionnel peut être utilisé pour assurer à la fenêtre un comportement déterministe. De la même façon, tout ordre supplémentaire qui ordonne totalement le flux peut être utilisé pour lever l'ambiguïté ; par exemple l'ordre des identifiants ou l'ordre lot-temps.

3.2.3.3 Utilisation des fenêtres en *DiSPL*

En pratique, l'implémentation d'un service continu est responsable de la création des fenêtres et de leur association à chaque port d'entrée. Un port d'entrée peut être associé à une seule fenêtre, qui est appliquée sur tous les flux connectés à ce port. *DiSPL* offre des fonctions primitives pour définir des fenêtres sur les ports d'entrées ; l'utilisation de ces fonctions étant réservée à la section d'initialisation :

```
(set-window <nom du port>
    (window <expression de la fenêtre, voir instruction window>)
)
(get-window <nom du port>)
(unset-window <nom du port>)
```

Une fois une fenêtre associée à un port donné, les primitives *get-new-items*, *get-next-item* et *get-next-items* lisent l'ensemble fini produit par la fenêtre et non plus les éléments lus depuis le flux. Les deux types de fenêtres disponibles sont les fenêtres positionnelles et les fenêtres temporelles, déclarées comme suit :

```
; fenêtre positionnelle
(window (type "position")
    (start-point <point de départ>)
    (end-point <point d'arrivée>)
    <stratégie de mise à jour>
    <stratégie de déclenchement>
)
; fenêtre temporelle
(window (type "time")
    (size <taille physique de la fenêtre>)
    (start-point <point de départ>)
    (end-point <point d'arrivée>)
    <stratégie de mise à jour>
    <stratégie de déclenchement>
)
```

La stratégie de mise à jour correspond au prédicat de mise à jour α dont les différentes formes sont celles présentées dans la Table 3.6 :

```
(update <"if-new" ou "on-change">)
(update ; mise à jour périodique
    (periodic <valeur de la période>)
)
(update ; mise à jour au changement de valeur d'attribut
    (on-value-change <liste des attributs>)
)
```

Les points de départ et d'arrivée correspondent à l'expression de la fonction d'évaluation β, conditionnée par la taille logique et la taille physique de la fenêtre :

```
(start-point <valeur fixe ou variable>)         ; point fixe
(start-point (now))                              ; point mobile
(start-point (+ (now) <constante ou variable>)) ; point mobile relatif
```

Enfin, la stratégie de déclenchement correspond au prédicat de déclenchement δ dont les différentes formes sont celles présentées dans la Table 3.7 :

```
(report <"if-change", "if-closed", "if-full" ou "if-not-empty">)
(report (periodic <valeur de la période>))
```

3.2.4 Opérateur *streamer*

Dans notre approche, les *streamers* sont des opérateurs de traitement de flux utilisés dans deux cas de figure :

- générer des flux à partir d'un stockage ;
- générer des flux à partir d'un historique de flux.

Définition 3.69 *Streamer* Un *streamer* est un opérateur qui transforme un ensemble fini X de n-uplets $x_1, \cdots, x_n$ en flux S. Pour cela, un *streamer* construit un ordre pour X et diffuse les éléments nouvellement ordonnés à un débit fixé.

L'utilisation d'un *streamer* pour construire un flux nécessite de spécifier le schéma respecté par les n-uplets de X. Si les n-uplets ne respectent pas une même structure, le flux produit est naturellement non structuré (sans schéma).

Définition 3.70 Fonction de construction d'ordre Une *fonction de construction d'ordre* $\triangleright\colon X \longrightarrow \mathcal{R}$ est une fonction qui associe chaque élément d'un ensemble fini X à un élément d'un ensemble ordonnant $\mathcal{R}$.

Étant donné un ensemble fini d'éléments respectant un schéma donné, nous introduisons les *streamers* suivants :

Définition 3.71 *Streamer* aléatoire Le *streamer aléatoire* diffuse les éléments de X dans un ordre quelconque non déterministe, dépendant de l'implémentation.

Définition 3.72 *Streamer* par attribut Le *streamer par attribut* utilise une fonction de construction d'ordre basée sur un attribut a présent dans chaque n-uplet de X. Selon le type de l'attribut, ces *streamers* peuvent construire des flux temporels ($\triangleright\colon X \longrightarrow \mathbb{T}$) ou des flux identifiés ($\triangleright\colon X \longrightarrow \mathbb{K}$).

Chapitre 4

Exécution de services distribués dans l'Internet des objets

Pour permettre le déploiement et l'exécution des applications orientées flux basées sur notre modèle de données et notre modèle de calcul, nous présentons *Dioptase*, un intergiciel installé sur les objets et gérant l'exécution des services qui composent les applications.

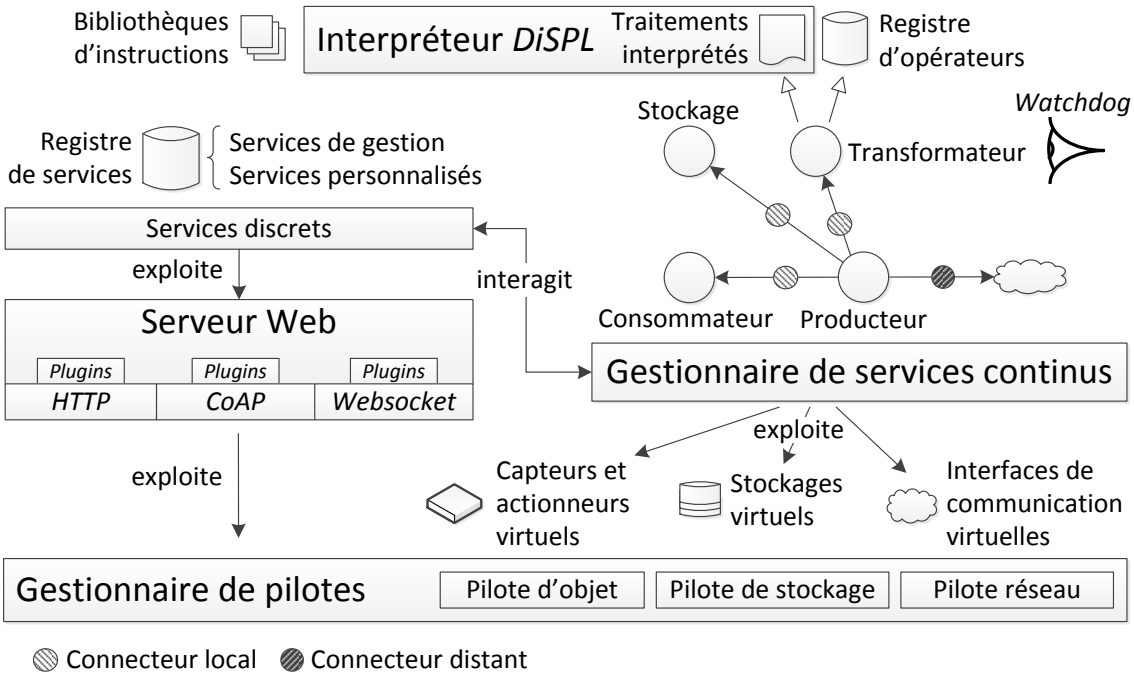


Figure 4.1 – Architecture de l'intergiciel *Dioptase*.

Dioptase implémente les concepts que nous avons introduits dans notre modèle de données (représentation des flux) et notre modèle de calcul (architecture orientée service pour le traitement continu). *Dioptase* est un intergiciel pouvant être déployé à tous les niveaux de l'Internet des objets, transformant chaque appareil en agent potentiellement autonome capable de communiquer directement avec ses pairs et d'héberger dynamiquement des services continus.

4.1 Architecture et implémentation de *Dioptase*

Notre prototype est développé en *Java* et déployé sur plusieurs types d'appareils aux capacités hétérogènes. Le choix de *Java* est motivé (i) par les avancées réalisées sur les machines virtuelles *Java* pour systèmes embarqués [165], (ii) l'existence des *Sun SPOT*, une technologie de capteurs sans fil programmables en *Java*, et (iii) le grand nombre de systèmes d'exploitation compatibles avec *Java*, nous permettant dès lors de travailler directement avec des systèmes très différents ; ordinateurs de bureau, téléphones mobiles, systèmes embarqués, etc. Il est à noter que l'utilisation de *Java* n'est en aucun cas un prérequis pour l'implémentation de *Dioptase*, des versions allégées pouvant être implémentées dans d'autres langages ; par exemple pour cibler d'autres familles de systèmes d'exploitation pour capteurs sans fil.

4.1.1 Composants de *Dioptase*

La Figure 5.4 présente une vue globale de l'architecture de *Dioptase*. Tous les composants logiciels présentés sur cette figure sont conçus pour être exécutés directement sur les objets. Pour pouvoir fonctionner sur un grand nombre d'objets différents, *Dioptase* est conçu de façon modulaire, permettant ainsi aux développeurs de composer des versions plus ou moins complexes de l'intergiciel en fonction des objets ciblés.

4.1.1.1 Pilotes

Tout d'abord, pour gérer les spécificités propres aux objets, les fonctionnalités de bas niveau sont découplées du reste de l'intergiciel au moyen de pilotes (*drivers*). Ces pilotes sont chargés au démarrage de *Dioptase* et sont utilisés par l'ensemble des autres modules, fournissant une interface générique pour l'accès aux fonctionnalités physiques de l'objet : la communication réseau, la lecture des capteurs, le contrôle des actionneurs et l'accès aux espaces de stockage persistants. En pratique, trois pilotes doivent être implémentés pour chaque plateforme, typiquement par le constructeur de celles-ci :

— Le *pilote d'objet*, qui encapsule (i) la logique de lecture des capteurs, (ii) la logique de contrôle des actionneurs et (iii) la logique d'accès aux métadonnées de l'objet.

— Le *pilote de stockage*, qui encapsule la logique de lecture-écriture dans les espaces de stockage disponibles.

— Le *pilote réseau*, qui encapsule la logique d'accès aux différents réseaux auxquels l'objet est connecté.

Les interfaces des trois pilotes sont détaillées dans l'Annexe 2. Une fois ces pilotes démarrés, *Dioptase* peut manipuler les *capteurs virtuels*, les *actionneurs virtuels*, les *stockages virtuels* et les *interfaces de communication virtuelles* de façon à gérer l'accès, le stockage et la diffusion des données mesurées ou traitées.

Définition 4.1 Capteur virtuel Un *capteur virtuel* est un composant logiciel fourni par le pilote d'objet pour exposer une source de données dont les valeurs changent au cours du temps.

Au-delà des capteurs physiquement intégrés à l'objet, le développeur peut intégrer n'importe quelle source d'information à *Dioptase* sous la forme de capteur virtuel ; par exemple l'accès à la quantité de mémoire consommée, un service de mesure externe ou encore des informations saisies par l'utilisateur dans un champ d'interface graphique.

Définition 4.2 Actionneur virtuel Un *actionneur virtuel* est un composant logiciel fourni par le pilote d'objet pour exposer une ou plusieurs actions permettant d'interagir avec le monde physique. Chaque action est encapsulée sous la forme d'un *service d'action*.

Définition 4.3 Service d'action Un *service d'action* est caractérisé par un nom d'action et des paramètres d'entrée (noms, types, domaines, optionnels ou non). Lorsqu'invoqué, un service d'action produit éventuellement une valeur résultat.

Définition 4.4 Stockage virtuel Un *stockage virtuel* est un composant logiciel fourni par le pilote de stockage pour accéder aux espaces de stockages.

Définition 4.5 Interface de communication virtuelle Une *interface de communication virtuelle* est un composant logiciel fourni par le pilote réseau pour échanger des données avec d'autres appareils de manière asynchrone.

4.1.1.2 Gestionnaire de services continus

Le *gestionnaire de services continus* est un composant logiciel majeur de *Dioptase*, dont le rôle est d'orchestrer l'exécution des différentes instances de services continus qui sont déployées sur l'objet. Spécifiquement, les responsabilités du gestionnaire de services continus sont les suivantes :

- instancier, contrôler et détruire les services continus qui s'exécutent sur l'objet ;
- collecter les mesures depuis les capteurs virtuels, à intervalle régulier ou lorsqu'un évènement survient, et les fournir aux producteurs ;
- gérer le cycle de vie des transformateurs ;
- gérer les *connecteurs* qui assurent la circulation des éléments entre les services continus reliés par des flux (voir la Section 4.1.2 pour le détail des connecteurs).

Au fil de l'exécution des transformateurs, un *watchdog* analyse comment les ressources matérielles de l'objet sont consommées. Lorsque les ressources mémoire et *CPU* viennent à manquer, le *watchdog* peut prendre la décision de détruire les transformateurs, conformément à des politiques fournies par les développeurs ou par les administrateurs de l'objet :

Définition 4.6 Politique de gestion de ressource Une *politique de gestion de ressource* exprime comment la consommation d'une ressource matérielle doit être gérée, au niveau global ou au niveau d'un transformateur. Elle se compose d'une *ressource observée*, d'une *stratégie d'analyse*, d'une *condition de surcharge* et d'une *stratégie de réduction de charge*.

Définition 4.7 Ressource observée La *ressource observée* par une politique de gestion désigne la ressource concernée par cette politique :

- Ressource de l'objet : mémoire de l'objet, charge *CPU* de l'objet, énergie de l'objet.
- Ressource allouée à un transformateur : mémoire consommée par la section de travail, nombre d'instructions exécutées par la section de travail, durée d'exécution de la section de travail, taille de l'état interne.

Définition 4.8 Stratégie d'analyse La *stratégie d'analyse* est une fonction qui à un instant t détermine quel sera l'état de la ressource à un instant $t + k$ étant donné une succession d'états passés d'une ressource.

Définition 4.9 État d'une ressource L'*état d'une ressource (temps t, valeur v)* correspond à sa valeur mesurée à un instant t : mémoire consommée, charge *CPU*, taille, durée, énergie restante.

Dioptase implémente deux stratégies d'analyse d'une ressource. Soient $(t_1, v_1), \cdots, (t_n, v_n)$ les n états passés d'une ressource :

— *Analyse constante* : la valeur v_{n+1} à t_{n+1} est prédite égale à la moyenne, le minimum ou le maximum des valeurs $v_1, \cdots, v_n$.

— *Analyse prédictive* : la valeur v_{n+1} à t_{n+1} est prédite par une fonction $f : v_{n+1} = f(t_{n+1})$. Typiquement, f est une régression linéaire des états passés.

Définition 4.10 Condition de surcharge La *condition de surcharge* permet de déterminer, à partir des résultats d'une stratégie d'analyse, quand appliquer une stratégie de réduction de charge.

Dioptase implémente les conditions de surcharge comme des seuils à ne pas dépasser. Dans le cas d'une analyse prédictive, le seuil peut être défini pour le temps présent $f(t)$ ou pour une prédiction future $f(t + k)$.

Définition 4.11 Stratégie de réduction de charge La *stratégie de surcharge* décrit quelles actions doivent être réalisées pour réduire la charge relative à la ressource observée.

Pour réduire la charge, *Dioptase* détruit des transformateurs en respectant l'une des stratégies suivantes :

— destruction aléatoire.

— destruction des plus anciens ou des plus récents en premiers ;

— destruction par ordre de priorité d'exécution ;

— destruction des plus gros consommateurs de ressources.

4.1.1.3 Implémentations par défaut des services continus

Les développeurs peuvent implémenter les producteurs, les consommateurs, les stockages et les transformateurs en étendant directement les interfaces logicielles fournies par *Dioptase* pour chaque type de service continu (voir Chapitre 3). Ces services continus sont alors compilés et déployés avec *Dioptase* sur les objets. Cependant, *Dioptase* fournit des implémentations par défaut pour chaque famille de service continu, de façon à simplifier l'écriture d'applications orientées flux.

Définition 4.12 Implémentation par défaut des producteurs L'implémentation par défaut des producteurs prend en paramètre un capteur virtuel et diffuse les valeurs mesurées sur sa sortie standard.

Définition 4.13 Implémentation par défaut des stockages L'implémentation par défaut des stockages prend en paramètre un type de stockage virtuel et une taille maximale. Cette implémentation possède un seul port d'entrée qui accepte une infinité de flux de n'importe quel type. Pour chaque flux S connecté sur ce port, le stockage crée (i) un sous-espace dans lequel sont stockés les éléments de S et (ii) un port de sortie dont le schéma est celui de S.

Définition 4.14 Implémentation par défaut des consommateurs L'implémentation par défaut des consommateurs prend en paramètre un nom de service d'action. Cette implémentation possède un seul port d'entrée qui n'accepte qu'un seul flux dont les attributs doivent correspondre aux noms et aux types des paramètres du service d'action.

Définition 4.15 Implémentation par défaut des transformateurs L'implémentation par défaut des transformateurs prend en paramètre un traitement continu à exécuter, un ensemble de valeurs correspondant aux paramètres du traitement, un contrat et un éventuel état initial. Les ports d'entrée et de sortie sont alors définis conformément au contrat fourni.

Un traitement compilé peut être implémenté soit en code natif, soit au moyen du *DSL* de traitement de flux *DiSPL* interprété directement par *Dioptase*. En pratique, les traitements compilés sont moins flexibles que les traitements interprétés, mais sont plus performants du fait de l'utilisation de code natif. À l'inverse, les traitements interprétés sont plus flexibles et peuvent être déployés à tout moment par les développeurs.

Définition 4.16 Traitement compilé Un *traitement compilé* est un traitement continu écrit dans l'un des langages natifs de la plateforme et interfacé directement avec *Dioptase*. Un traitement compilé possède un nom, qui permet de l'associer à un transformateur à posteriori.

Un exemple de traitement compilé est présenté sur la Figure 4.2 ; ce code source implémente le programme de comptage présenté comme exemple dans le Chapitre 3 (Section 3.2.2.3). *Dioptase* fournit un ensemble de traitements compilés qui implémentent les opérateurs de traitement continu usuels pouvant être exécutés sur n'importe quel objet : sélection, projection, agrégation, etc. Au-delà de ces fonctionnalités minimales, *Dioptase* intègre un mécanisme de paquetage permettant la création de bibliothèques de traitements prêtes à l'emploi. Le *registre d'opérateurs* maintient la liste des différents traitements compilés et de leurs contrats.

Définition 4.17 Traitement interprété Un *traitement interprété* est un traitement continu écrit en *DiSPL*, interprété dynamiquement par *Dioptase*. Le programme *DiSPL* peut être déployé sur n'importe quel objet à posteriori.

Comme présenté dans la Figure 4.3, les programmes *DiSPL* sont convertis en *arbre de syntaxe abstraite* (*AST*) par l'analyseur syntaxique. Une fois construit, cet arbre est

```java
public class CountProcessor extends AbstractProcessor
{
    public CountProcessor()
    {
        super(new ContractBuilder("count-processor") // construction du contrat
            .singleStreamInput("main-port")
            .output("count-result", new SchemaOutputRule(
                new Schema("count",
                    SemanticTypes.None, Units.None,
                    PrimitiveTypes.Integer)
            )).build()
        );
    }
    @Override
    public void initialize() // section d'initialisation
    throws ExecutionException
    {
        getState().set("count", IntValue.recycle(0)); // création du compteur
    }
    @Override
    public void process() // section de travail
    throws ExecutionException
    {
        // récupération des éléments en attente
        List<StreamItem> items = getNewItems("main-port");
        if(items.size() > 0)
        {
            // récupération et mise à jour de la valeur du compteur
            IntValue countWrapper = (IntValue) getState().get("count");
            Integer count = countWrapper.asInteger();
            count = count + items.size();
            countWrapper.set(count);

            StreamItem item = new StreamItem(Time.now(), "count", count);
            getStandardOutput().write(item); // écriture du compte sur la sortie standard
        }
    }
}
```

Figure 4.2 – Exemple de traitement écrit en *Java* et compilé avec *Dioptase*.

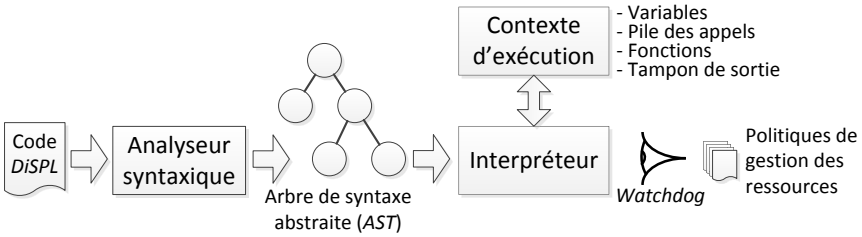


Figure 4.3 – Architecture de l'interpréteur *DiSPL*.

associé à un *contexte d'exécution*, utilisé pour stocker des informations telles que les variables ou la pile des appels. Piloté par le transformateur, l'interpréteur *DiSPL* exécute chaque section du programme et stocke les variables globales dans l'état interne du transformateur.

4.1.1.4 Serveur Web embarqué

L'accès aux opérations fournies par les services discrets et aux flux produits par les services continus est implémenté au moyen de services Web de type *REST*. Pour ce faire, *Dioptase* embarque un client et un serveur Web tout-en-un, optimisé pour fonctionner avec des ressources limitées :

- Le serveur Web est divisé en modules, chaque module implémentant un protocole de service : services *HTTP*, services *CoAP* et *websockets*. À la compilation de *Dioptase*, les modules peuvent être liés indépendamment de façon à ne conserver que ceux requis par le développeur.
- Chaque implémentation de protocole est composée d'un corpus minimal de fonctionnalités, suffisant pour implémenter les scénarios usuels d'invocation de service et de diffusion de flux.
- Chaque implémentation de protocole s'accompagne de *plugins* qui implémentent les fonctionnalités supplémentaires du protocole : compression des données, chiffrement, etc. Ces plugins peuvent eux aussi être liés à la compilation de *Dioptase*.

La liste des fonctionnalités de base et des *plugins* fournis pour chaque type de protocole est présentée dans l'Annexe 3.

Définition 4.18 Service de gestion Un *service de gestion* est un service discret permettant de gérer l'exécution de *Dioptase* et des services continus qu'il héberge.

Les services de gestion implémentés par *Dioptase* permettent, notamment :

- déployer et contrôler les services continus (voir Section 4.1.3 pour les détails du déploiement) ;
- de récupérer des informations directement au niveau des pilotes (accès à des données ou récupération d'une mesure) ;
- d'accéder aux métadonnées de l'objet (capteurs, actionneurs, matériel, position, niveau de batterie, opérateurs disponibles, etc.) ;
- de lister les services continus qui sont en cours d'exécution (*URI* de chaque flux d'entrée et de sortie, contrats, schémas, etc.).

Les services de gestion disponibles et leurs paramètres sont présentés en détail dans l'Annexe 4.

4.1.2 Connecteurs et protocoles de transport de flux

Comme montré sur la Figure 4.4, l'accès aux données produites par des services continus se décompose en deux étapes :

Définition 4.19 Requête d'accès au flux Une *requête d'accès au flux* est un message émis par un récepteur de flux r à destination d'un émetteur de flux s pour initier l'échange de données. La requête d'accès au flux est acheminée au moyen d'un *protocole d'accès*.

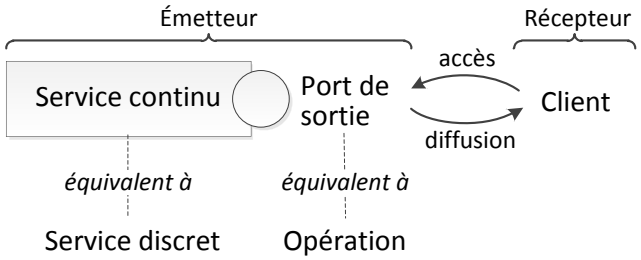


Figure 4.4 – Accès à un flux.

Définition 4.20 Canal de diffusion de flux Un *canal de diffusion de flux* est une transmission d'éléments de flux qui s'établit d'un émetteur de flux s vers un récepteur de flux r après que s ait reçu une requête d'accès au flux de la part de r. Le canal de diffusion est maintenu par un *protocole de diffusion de flux*.

La séparation de ces deux aspects permet notamment de spécifier des implémentations où l'accès aux flux et la diffusion de flux sont assurés par deux protocoles différents. En effet, une technique couramment utilisée pour l'intégration des flux de données dans *SOA* consiste à utiliser un canal séparé pour le transport du flux [153] ; typiquement une connexion *TCP* distincte. Les services sont alors utilisés pour récupérer l'adresse et le port du canal (requête d'accès) et, éventuellement, pour le contrôle du flux par la suite ; par exemple la mise en pause d'un flux multimédia.

Dans notre cas, les protocoles utilisés sont :

— *HTTP* et *CoAP* [77] pour les requêtes d'accès aux flux ;

— *HTTP streaming*, *Web hook*, *websocket* [150] et *CoAP* pour les canaux de diffusion de flux.

Concrètement, ce sont les *connecteurs* qui abstraient la communication entre les services continus. Un connecteur implémente un protocole de requête d'accès au flux et un protocole de diffusion de flux et prend en charge les différentes problématiques de transport.

Définition 4.21 Connecteur Un *connecteur* est un composant logiciel qui connecte un port de sortie d'un service continu vers un port d'entrée d'un autre service continu.

Pour peu que les contraintes sur ports d'entrée soient satisfaites, il est possible de connecter les sorties de n'importe quel composant sur le port d'entrée d'un autre composant. Cette connexion est établie par le connecteur qui gère alors tous les aspects liés au transport, à l'adaptation et à la présentation des données. En pratique nous pouvons classer les connecteurs en deux familles :

Définition 4.22 Connecteur local Un *connecteur local* gère la communication entre deux services continus qui s'exécutent au sein d'un même appareil et optimise le transport du flux en conséquence (appels locaux).

Définition 4.23 Connecteur distant Un *connecteur distant* prend en charge la communication entre deux services continus qui s'exécutent sur des objets différents. Les connecteurs distants possèdent une partie émetteur et une partie récepteur pour l'envoi et la réception des éléments du flux.

Dioptase implémente trois connecteurs distants : le connecteur *HTTP*, le connecteur *CoAP* et le connecteur *websocket*. Chaque connecteur peut être utilisé en fonction des besoins ; par exemple le connecteur *CoAP* est parfaitement adapté pour les objets limités en ressources matérielles tandis que le connecteur *websocket* est particulièrement utile pour exposer des flux à destination d'applications Web dynamiques.

4.1.2.1 Connecteur *HTTP*

Le connecteur *HTTP* est basé sur le protocole *HTTP* et ses différentes spécialisations pour l'invocation de service et la diffusion de flux. Le protocole d'accès utilisé est celui des services *RESTful* et correspond à une requête *HTTP GET* sur un *URI* particulier :

— */<nom du service continu>* pour accéder à la sortie standard d'un service continu ;

— */<nom du service continu>/<nom du port de sortie>* pour accéder à l'un des ports de sortie d'un service continu ;

Le connecteur *HTTP* propose deux protocoles de diffusion de flux, respectivement basés sur les techniques *HTTP streaming* et *Web hooks* que nous avons présentés dans le Chapitre 2. À titre d'exemple d'échange réalisé au travers du connecteur *HTTP*, considérons un producteur nommé *light3* dont le rôle est de fournir des mesures de luminosité. Le flux produit par la sortie standard de *light3* est un flux temporel dont les éléments ne possèdent qu'un attribut, nommé *light*. La capture réseau suivante présente la requête d'accès à la sortie standard de *light3* et la diffusion de deux éléments au moyen de la technique *HTTP streaming* :

`GET /light3?mode=stream HTTP/1.1\r\n ...\r\n\r\n`	Requête d'accès reçue pour *light3*. Le protocole de diffusion demandé est *HTTP streaming* (*mode=stream*).
`Transfer-Encoding: chunked\r\n Content-Type: application/json\r\n ...\r\n\r\n`	La réponse *HTTP* est configurée en mode bloc [82] (un élément = un bloc) et les éléments sont sérialisés en *JSON*.
`1d\r\n {"t":566218800,"light":226.3}\r\n`	Premier élément en *JSON*, avec *t* le timestamp et *light* le nom de l'attribut.
`1e {"t":566218860,"light":201.08}\r\n`	Second élément en *JSON*.
`...`	
`0\r\n\r\n`	La connexion n'est pas fermée par le serveur tant que le flux n'est pas terminé. La fin de la réponse *HTTP* est indiquée par un bloc vide [82].

La capture réseau suivante présente la requête d'accès au port standard de *light3* et la diffusion de deux éléments au moyen des *Web hooks* :

```
GET /light3?mode=hook&hook=hook1
HTTP/1.1\r\n
...\r\n\r\n
HTTP/1.1 200 OK
...
```

Requête d'accès reçue pour *light3*. Le protocole de diffusion demandé est *Web hook* (*mode=hook*) et un hook nommé *hook1* est précisé. Pour chaque élément produit par *light3*, le connecteur de l'émetteur enverra une requête *GET* au récepteur sur l'*URI /hooks/hook1*.

```
GET /hooks/hook1?t=566218800
&light=226.3 HTTP/1.1\r\n
...\r\n\r\n
HTTP/1.1 200 OK\r\n
...
```

Requête de l'émetteur vers le récepteur. Le premier élément est encodé dans la chaîne d'interrogation de l'*URI*, avec *t* le timestamp et *light* le nom de l'attribut.

```
GET /hooks/hook1?t=566218860
&light=201.08 HTTP/1.1\r\n
...\r\n\r\n
HTTP/1.1 200 OK
...
```

Requête de l'émetteur vers le récepteur, avec le second élément.

4.1.2.2 Connecteur *CoAP*

Les exemples de flux que nous venons de présenter sont des flux *HTTP* qui, en pratique, ne sont pas adaptés pour les appareils fortement restreints en ressources, du fait de la verbosité élevée du protocole *HTTP*, ce même une fois optimisé. Pour ces environnements particuliers, le protocole *CoAP* [77] peut être utilisé pour le transport de flux bien que, à notre connaissance, ce cas d'utilisation n'ait pas été étudié en pratique dans la littérature du fait de la nature récente du protocole.

CoAP prend en charge les mêmes méthodes *RESTful* qu'*HTTP*, mais ne permet pas d'émuler le comportement d'*HTTP streaming*, étant donné l'utilisation d'*UDP* comme couche de transport (pas de connexion). Toutefois, la diffusion de flux par *Web hooks* peut être reproduite avec *CoAP* moyennant les adaptations suivantes :

— réordonner les messages à la réception étant donné qu'*UDP* n'offre aucune garantie de respect de l'ordre d'émission ;

— utiliser les mécanismes d'acquittement (messages de type *confirmables*) et de retransmission spécifiés par le protocole [77].

À cette fin, le connecteur *CoAP* fournit deux modes de fonctionnement relatifs au respect de l'ordre et à la tolérance aux pertes :

Définition 4.24 Mode ordonné Lorsqu'il est en *mode ordonné*, le connecteur *CoAP* réordonne les éléments de flux à la réception avant de les transmettre aux services continus concernés.

Définition 4.25 Mode fiable Lorsqu'il est en *mode fiable*, le connecteur *CoAP* utilise les mécanismes d'acquittement et de retransmission spécifiés par le protocole.

Les deux modes peuvent être exploités conjointement, les quatre combinaisons ayant des comportements différents décrits en détail dans la Table 4.1.

	Fiable et ordonné	**Fiable et non ordonné**	**Non fiable et ordonné**	**Non fiable et non ordonné**
Émetteur	messages *confirmables*	messages *confirmables*	*fire&forget*	*fire&forget*
Récepteur	maintient une table des éléments manquants, route les groupes continus d'éléments ordonnés	route les éléments dès leur réception	route les éléments dès leur réception si leurs identifiants sont supérieurs à celui du dernier message reçu	route les éléments dès leur réception
Échec de l'émetteur	nombre maximal de retransmissions dépassé pour un élément non acquitté	nombre maximal de retransmissions dépassé pour un élément non acquitté		
Échec du récepteur	temps d'attente maximal dépassé pour un élément manquant			

Table 4.1 – Comportement du connecteur *CoAP*.

4.1.3 Déploiement et contrôle des services continus

Les services de gestion permettent aux développeurs et aux autres objets de contrôler les services continus qui s'exécutent sur un objet. Ils sont implémentés sous la forme de services *RESTful HTTP* ou *CoAP* dont la liste détaillée est donnée en Annexe 4.

En ce qui concerne le déploiement, le service de gestion */services* permet d'instancier les implémentations de services continus qui ont été compilées avec *Dioptase*, en fournissant le nom du service continu et les paramètres requis. Par ailleurs, des services de gestion spécialisés permettent de déployer les implémentations par défaut des quatre familles de services continus, en fournissant les paramètres suivants :

— *Déploiement d'un producteur* : requiert le nom du capteur virtuel à utiliser comme source de données.

— *Déploiement d'un stockage* : requiert le type de l'espace de stockage virtuel à créer, une taille maximale pour le stockage et une liste de flux à connecter à chaque port d'entrée.

— *Déploiement d'un consommateur* : requiert le nom du service d'action à utiliser et une liste de flux à connecter à chaque port d'entrée.

— *Déploiement d'un transformateur* : requiert le nom du traitement compilé ou le code *DiSPL* à exécuter, l'état interne de départ, les paramètres requis par le traitement continu et une liste de flux à connecter à chaque port d'entrée.

Les services de gestion chargés du déploiement font appel au gestionnaire de services continus pour l'instanciation du service continu et la connexion de ses ports d'entrée aux ports de sortie des services qui produisent les flux de données requis. Ce mode

d'interaction, dit *pull*, se justifie par notre volonté de rendre les objets plus autonomes ; ainsi, un objet ne peut pas recevoir un flux sans l'avoir précédemment accepté.

À titre d'exemple, considérons le cas où un client souhaite déployer, sur un objet A, le programme de comptage écrit en *DiSPL* que nous avons présenté au Chapitre 3. Le client souhaite que le port d'entrée *main-port* du transformateur soit connecté à la sortie standard d'un producteur appelé *light3*, déjà déployé sur un second objet B. La capture réseau suivante présente la requête envoyée à A par le client pour déployer le programme *DiSPL* et le connecter à *light3* :

`POST /processors HTTP/1.1\r\n`
`Content-Type: multipart/form-data;`
`boundary=fyrdm2\r\n`
`...\r\n\r\n`
	Requête pour le déploiement d'un nouveau transformateur. Spécifie que les paramètres sont encodés en mode *multipart*, un format classique où chaque paramètre est séparé par un délimiteur (*boundary*) [166].

`--fyrdm2\r\n`
`Content-Disposition: form-data;`
`name="id"\r\n\r\n`
`some-processing\r\n`
	Premier paramètre, *id*, qui spécifie le nom désiré pour le transformateur.

`--fyrdm2\r\n`
`Content-Disposition: form-data;`
`name="code"\r\n\r\n`
`<some DiSPL code>\r\n`
	Second paramètre, *code*, qui spécifie le code *DiSPL* à exécuter.

`--fyrdm2\r\n`
`Content-Disposition: form-data;`
`name="main-port"\r\n\r\n`
`http://173.194.34.24:9000/light3\r\n`
	Chaque nom de port d'entrée est un paramètre potentiel spécifiant l'*URI* de chaque flux (une par ligne) à connecter à ce port. Ici, *main-port* est le port concerné, le schéma et le domaine de l'*URI* fourni spécifiant qu'un connecteur *HTTP* doit être utilisé pour accéder au flux produit par la sortie standard de *light3*.

`--fyrdm2\r\n`
`Content-Disposition: form-data;`
`name="autostart"\r\n\r\n`
`true\r\n`
	Quatrième paramètre, *autostart*, qui spécifie que le transformateur doit démarrer immédiatement.

`--fyrdm2--`
`HTTP/1.1 200 OK\r\n`
`...`
	Fin du corps de requête.

Une fois démarré, un service continu peut être arrêté à tout moment par le client, en envoyant une requête *DELETE* au service de gestion */services*.

4.1.4 Exemple d'application écrite avec *Dioptase*

Comme nous l'avons déjà évoqué, une application orientée flux est composée de services continus répartis sur un ensemble d'objets équipés de *Dioptase*. Au fur et à mesure de son exécution, une application peut déployer de nouveaux services continus en fonction de ses besoins et les composer sous la forme de tâches décrites par un graphe

logique. L'exemple d'application que nous considérons ici est celui d'un *serveur d'exécution de requête* offrant les fonctionnalités suivantes :

— exprimer des requêtes de traitement de flux dans un langage de requête proche de *SQL*, comme c'est le cas pour *TinyDB* [42] ;

— transformer ces requêtes en graphe logique et exécuter celui-ci.

Pour la simplicité de l'exemple, nous considérons uniquement des requêtes basées sur un nombre réduit de traitements : la sélection, la projection, les agrégations et l'équijointure. La grammaire *ABNF* [167] de notre langage de requête, baptisé *DiSQL*, est la suivante :

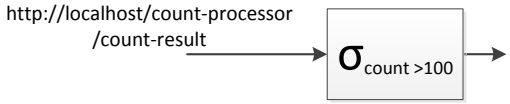
```
request = (simple-select / join)
simple-select = "SELECT " attributes " FROM " sources [" WHERE " predicates]
join = "SELECT " attributes " FROM " source " INNER JOIN " source "
            USING " attribute-name
attributes = attribute *("," SP attribute)
attribute = (aggregated-attribute / natural-attribute)
aggregated-attribute = aggregation-operator "(" natural-attribute ")"
aggregation-operator = ("COUNT" / "AVG" / "MAX" / "MIN" / "SUM" / "PRODUCT")
natural-attribute = ("*" / [qualifier "."] attribute-name)
qualifier = alphanum
attribute-name = alphanum
sources = source *("," SP source)
source = (source-uri / [source-name "/"]source-port) [SP qualifier]
source-name = alphanum
source-uri = uri
source-port = alphanum
predicates = predicate *(SP boolean-operator SP predicate)
boolean-operator = ("OR" / "AND")
predicate = operand SP operator SP operand
operator = ("=" / "!=" / "<" / ">" / "=" / "=")
operand = (attribute / value)
value = (string / 1*DIGIT)
string = DQUOTE *VCHAR DQUOTE
alphanum = ALPHA *(ALPHA / DIGIT)
```

Le langage permet d'exprimer différentes requêtes, dont quelques exemples sont présentés dans la Table 4.2. Les différents traitements pris en charge par les requêtes appartiennent à la bibliothèque standard de *Dioptase* et sont donc disponibles sur tous les objets. Les transformateurs qui constituent la requête sont exécutés sur l'objet qui héberge le serveur d'exécution de requête et connectés aux flux produits par les services continus locaux ou distants spécifiés dans les clauses *FROM* et *INNER JOIN*.

L'architecture de l'application est décomposée en trois parties : (i) un service de gestion, qui reçoit les requêtes *DiSQL* à exécuter, (ii) un *analyseur de requête*, qui construit un graphe logique à partir des requêtes *DiSQL*, et (iii) un *orchestrateur*, qui déploie localement les services continus qui constituent la requête et construit les connecteurs appropriés.

L'implémentation du service de gestion consiste à récupérer le code de la requête *DiSQL* parmi les paramètres reçus par le service, puis à utiliser l'analyseur syntaxique pour construire un graphe logique. Enfin, le service de gestion démarre l'exécution du graphe

`SELECT MAX(light) FROM light3` 	Recherche le maximum de l'attribut *light* d'un flux produit par la sortie standard d'un service continu local appelé *light3*
`SELECT * FROM` `http://173.194.34.24:9000/light3`	Récupère le flux produit par la sortie standard d'un service continu distant appelé *light3*

`SELECT count FROM`
`count-processor/count-result`
`WHERE count > 100`
 
	Sélectionne les valeurs supérieurs à 100 de l'attribut *count* d'un flux produit par le port *count-result* d'un service continu local *count-processor light3*

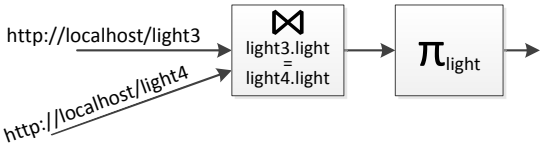
`SELECT light FROM light3 INNER`
`JOIN light4 USING (light)`
 
	Effectue une jointure sur deux flux de luminosité produits par les sorties standard de deux services continu locaux *light3* et *light4*. L'attribut utilisé pour l'équijointure est l'attribut *light*.

Table 4.2 – Quelques exemples de requêtes *DiSQL*.

logique et le client n'a plus qu'à se connecter au port approprié du dernier transformateur de la requête (puit) pour acquérir les résultats.

```
public class DiSQLRequestService extends AbstractManagementService
{
    @Override
    public void invoke(Request request, Response response)
    throws ServiceException
    {
        String disqlRequest = request.getParameter("disql");
        LogicalGraph graph = ...; // analyse syntaxique de la requête
        new SimpleOrchestrator().execute(graph);
        // le service répond en indiquant le graphe produit
        response.setBody(new JSONLogicalGraph(graph).toString());
    }
    @Override
    public String getPath()
    {
        return "/request-server"; // URI du service de gestion
    }
}
```

L'implémentation de l'orchestrateur consiste à déployer localement les services continus qui composent le graphe logique et à créer les différents connecteurs au moyen du gestionnaire de services continus.

```
public class SimpleOrchestrator extends AbstractOrchestrator
{
    @Override
    public void execute(LogicalGraph graph)
    {
        StreamServiceManager mgr = getStreamServiceManager();
        for(LogicalVertex vertex : graph)
        {
            Processor proc = mgr.deployProcessor("localhost", vertex.getName(),
                vertex.getParameters(), vertex.getInputURIs());
            mgr.startProcessor(proc);
        }
    }
}
```

4.1.5 Déploiement de *Dioptase* sur les objets

Une fois déployé sur les appareils, *Dioptase* offre les interfaces nécessaires pour déployer les services continus. Typiquement, déployer l'intergiciel sur un objet se fait en deux étapes :

Définition 4.26 Phase de personnalisation La *phase de personnalisation* correspond aux actions effectuées par les développeurs pour ajouter ou retirer des modules de *Dioptase*. À la fin de cette phase, les développeurs sont en possession d'une version personnalisée de l'intergiciel, adaptée aux objets ciblés.

Cette phase de personnalisation est réalisée par les développeurs en fonction des ressources matérielles des objets et des besoins applicatifs ; par exemple, si seuls les traitements compilés sont nécessaires, l'interpréteur *DiSPL* peut être retiré. En outre, cette phase permet au développeur d'implémenter ses propres modules, notamment pour prendre en charge les capacités spécifiques des objets ; par exemple, le décodage vidéo matériel.

Définition 4.27 Phase de déploiement La *phase de déploiement* correspond aux actions effectuées par les développeurs pour compiler la version personnalisée de *Dioptase* et la déployer sur les objets ciblés.

Concrètement, personnaliser *Dioptase* est une tâche relativement simple du fait de la modularité de l'intergiciel, mais qui requiert cependant des connaissances spécifiques à propos (i) des modules de *Dioptase* et (ii) des objets ciblés. Bien que la phase de personnalisation ne doive être réalisée qu'une seule fois pour une catégorie d'objets donnée, cela peut nécessiter un certain investissement en temps de la part du développeur. Toutefois, une grande partie de cette phase peut être simplifiée en distribuant des versions prépackagées et préconfigurées de *Dioptase*, précompilées en fonction des ressources matérielles disponibles sur une catégorie d'objets. Enfin, en ce qui concerne le développement de fonctionnalités spécifiques aux objets, les modules les plus utilisés pourront être partagés entre les développeurs au moyen de sites et de dépôts spécialisés. Dans le futur, ces modules pourraient être fournis par les fabricants eux-mêmes, tout comme les pilotes.

4.2 Discussion sur la protection de la vie privée

Comme nous en avons déjà parlé, la protection des données produites au sein de l'Internet des objets est une problématique d'importance croissante. Notre approche de protection des données privées produites par les objets appartenant aux utilisateurs se base sur deux propriétés :

1. l'autonomie des objets, capables d'interagir entre eux plutôt que de transférer les données à de grandes infrastructures tierces ;
2. le cloisonnement des réseaux d'objets sous la forme d'*espaces* dotés de mécanismes proches de la *révélation à la demande* [168].

Cette section discute les concepts relatifs au second point et les mécanismes permettant de protéger les données privées au sein des réseaux d'objets.

4.2.1 Espaces publics et privés, agrégats d'espaces

Définition 4.28 Espace Un *espace* est un groupe d'objets administrés par une entité unique. Un espace se compose : (i) d'objets passifs, (ii) d'objets actifs capables de mesurer et d'agir sur l'environnement, (iii) de terminaux pour l'accès aux services de mesure et d'action fournis par les objets, et (iv) d'une infrastructure locale à l'espace permettant aux différents appareils de communiquer entre eux : services locaux pour le nommage, l'adressage et la découverte d'appareils et de services.

Le réseau domotique d'un appartement est un exemple d'espace administré par son propriétaire, tout comme l'est un réseau d'entreprise géré par un service informatique interne. Concrètement, un espace est autonome, en cela que les éléments qui le constituent ne nécessitent pas d'interactions avec l'extérieur pour fonctionner, sauf dans des cas particuliers spécifiquement encadrés ; par exemple, le recours à des services tiers.

Les espaces peuvent être soit *privés*, comme un domicile ou un bureau, ou *publics*, comme une ville ou un parking, la distinction entre les deux étant la suivante :

Définition 4.29 Espace public Un espace est dit *public* lorsque les objets qui le constituent et les informations qui circulent dans celui-ci sont accessibles à tous, sans contrôle d'accès.

Définition 4.30 Espace privé Un espace est dit *privé* lorsque celui-ci est soumis à des politiques de contrôle d'accès aussi bien physiques que logicielles.

Une fois délimités, les espaces peuvent échanger les informations qu'ils produisent avec des entités extérieures, notamment des utilisateurs, des organisations et d'autres espaces.

Définition 4.31 Point d'échange Un *point d'échange* est une entité appartenant à un espace X et dont le rôle consiste à gérer comment d'autres espaces peuvent accéder aux informations et aux services offerts par les objets de X.

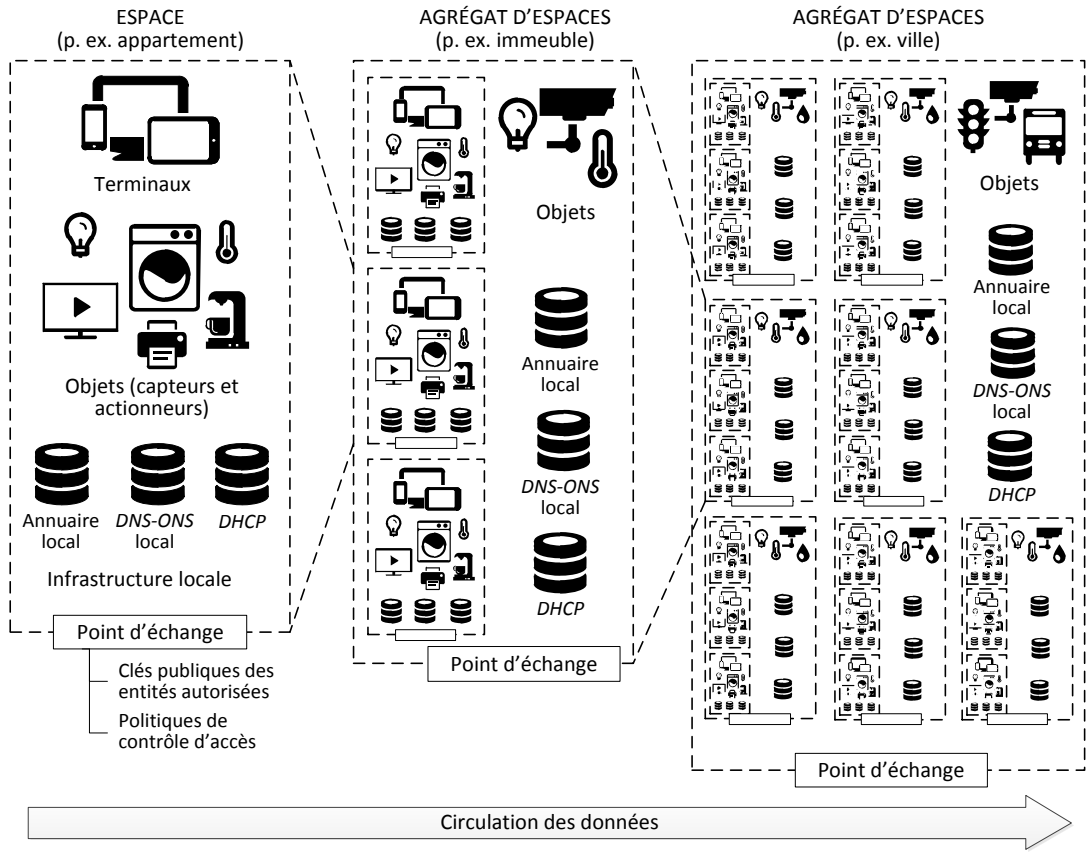


Figure 4.5 – Espaces et agrégats d'espaces.

Définition 4.32 Agrégat d'espaces Un *agrégat d'espaces* est un espace regroupant plusieurs espaces existants ; les échanges entre l'agrégat d'espace et les espaces agrégés étant gérés par les différents points d'échanges.

La Figure 4.5 illustre le concept d'agrégat d'espaces au travers de l'exemple des villes et des logements intelligents : un immeuble de logements est un espace privé qui (i) agrège plusieurs espaces privés plus petits (les appartements) et (ii) possède ses propres objets pour diverses tâches relatives à la gestion de l'éclairage, de l'accès physique (digicode) ou de la vidéosurveillance.

4.2.2 Description et application des politiques de contrôle d'accès

En pratique, les points d'échange sont chargés de gérer l'accès depuis l'extérieur aux informations et aux services mis à disposition par l'espace. Pour ce faire, ils maintiennent un ensemble d'informations permettant d'authentifier les agents extérieurs :

Définition 4.33 Agent extérieur Un *agent extérieur* est un appareil capable de prouver son identité auprès d'un point d'échange, typiquement grâce à un certificat (couple de clés asymétriques).

En pratique, les appareils appartenant à un utilisateur sont le plus souvent identifiés avec un certificat unique (identité de l'utilisateur). De la même façon, les appareils appar-

tenant à un espace sont le plus souvent identifiés avec un certificat commun à l'espace, bien que rien n'empêche chaque appareil de posséder un ou plusieurs certificats propres. En ayant connaissance au préalable de ces informations d'identification-authentification, l'administrateur d'un espace peut définir les politiques de contrôle d'accès aux données et aux services, soit pour des utilisateurs seuls ou des groupes d'utilisateurs indépendants, soit pour des espaces, c'est-à-dire pour l'ensemble des machines qui en font partie.

Définition 4.34 Politique de contrôle d'accès Une *politique de contrôle d'accès* est une règle qui porte sur l'accès à une *ressource privée* par un agent extérieur. À cette règle est associée un *contexte*, qui décrit les modalités d'accès à la ressource.

Définition 4.35 Ressource privée Une *ressource privée* est un service ou un flux de données mis à disposition par un objet à l'intérieur d'un espace. Un service est caractérisé par son *URI* tandis qu'un flux de données est caractérisé par son schéma et, notamment, son type sémantique.

Définition 4.36 Contexte d'accès à un service Le *contexte d'accès à un service* définit les modalités d'invocation d'un service, en spécifiant :

— les plages horaires pendant lesquelles l'agent extérieur peut invoquer le service ;

— le nombre d'invocations maximum pour un intervalle de temps fixé ;

— le temps d'attente minimal entre chaque invocation.

Définition 4.37 Contexte d'accès à un flux de données Le *contexte d'accès à un flux de données* définit les modalités de lecture d'un flux de données, en spécifiant :

— les plages horaires pendant lesquelles l'agent extérieur peut lire le flux ;

— les stratégies de réduction d'information.

Définition 4.38 Stratégie de réduction d'information Une *stratégie de réduction d'information* est un traitement appliqué à un flux de données dans le but de réduire la portée ou la quantité des informations qui y circulent.

Les différentes stratégies implémentées dans le prototype de *Dioptase* proviennent de la littérature sur la protection des données privées dans les réseaux de capteurs sans fil [169]. Elles sont notamment décrites dans la Table 4.3.

Une fois les politiques définies, le point d'échange agit en tant qu'intermédiaire entre l'espace et les requêtes extérieures. En effet, les politiques de contrôle d'accès ont cela d'intéressant qu'elles définissent les limites de l'espace virtuel, ces limites pouvant ensuite être associées aux limites physiques correspondantes de l'espace. Par exemple, un scénario de contrôle d'accès physique par badge peut être implémenté en exploitant directement (i) les identités stockées au niveau du point d'échange et (ii) des règles spécifiques pour la limitation des zones physiques auxquelles un utilisateur a accès.

Stratégie	Détails
Préagrégation	Un simple compteur électrique suffit pour déterminer si une personne se trouve chez elle et quelles sont ses activités grâce aux fluctuations de la consommation énergétique. La préagrégation des mesures sur une période de temps donnée gomme naturellement ces détails.
Présélection	Dans certains cas d'utilisations (surveillance, détection de pannes, etc.), seules certaines informations sont utiles ; par exemple, un pic de tension électrique. La présélection permet de filtrer ces informations à priori.
Discrétisation	Certains scénarios peuvent être implémentés sans qu'il soit nécessaire de diffuser des informations quantitatives. La discrétisation transforme des plages de valeurs en classes prédéfinies ; par exemple *silencieux*, *bruyant*, etc.
Perturbation	Lorsqu'une précision à $\pm x$ est suffisante, il n'est pas nécessaire de transmettre les informations exactes. Ce mode perturbe aléatoirement les valeurs dans un intervalle x fixé.
Sous-échantillon	Limite le nombre de mesures pouvant être obtenues dans un intervalle de temps donné en spécifiant soit une fréquence d'échantillonnage maximale, soit le temps d'attente minimum entre deux mesures.
Brouillage	Mode de perturbation et de sous-échantillonnage qui introduit, en outre, des mesures aléatoires.

Table 4.3 – Stratégies de réduction de la portée des informations privées.

Définition 4.39 Requête inter-espace Une *requête inter-espace* est une requête d'accès à une ressource, émise par un appareil x d'un espace X à destination d'un appareil y d'un espace Y. Une requête inter-espace peut être soit (i) une requête d'invocation de service soit (ii) une requête d'accès à un flux.

La propagation d'une requête inter-espace d'un appareil x d'un espace X vers un appareil y d'un espace Y est décrite par l'Algorithme 4.1

Les différentes politiques que nous venons de présenter peuvent être complexes à spécifier en pratique, notamment pour les utilisateurs finaux. Une simplification future pourrait consister à utiliser des ensembles de politiques prédéfinies, associées à des niveaux de confiance. Grâce à ces profils, l'administrateur de l'espace n'aurait alors plus qu'à annoter les différents utilisateurs, organisations et espaces avec un niveau de confiance, qui correspondrait à un ensemble de politiques prêtes à l'emploi.

4.2.3 Autres mécanismes de protection

Outre les politiques réduisant la quantité d'information que l'on peut extraire des mesures, les agrégats d'espaces peuvent atténuer le caractère individualisant des informations. En effet, un agrégat d'espace peut prétraiter les informations issues de tous les espaces qui le composent et les agréger de façon à ce qu'il ne soit plus possible de déterminer les composantes individuelles de ces données. Par exemple, si l'on considère

Algorithme 4.1 Propagation d'une requête inter-espace

Entrées : L'objet x de l'espace X souhaite interagir avec l'objet y de l'espace Y

1: x construit une requête inter-espace Q à destination de y
2: x transmet Q au point d'échange X_p de l'espace X
3: X_p transmet Q au point d'échange Y_p de l'espace Y
4: Y_p analyse les certificats connus et recherche le certificat de x ou de X_p
5: **si** Y_p possède un certificat C_x pour x ou X_p **alors**
6: Y_p recherche les politiques associées à C_x
7: **si** x ou X_p possède l'autorisation d'accéder à la ressource demandée sur y **alors**
8: Y_p transmet Q à y
9: y traite Q et produit un résultat R (soit un flux de données, soit un ensemble fini)
10: y transmet R vers Y_p
11: Y_p applique les stratégies de réduction d'information, si nécessaire
12: Y_p transmet le résultat modifié R' à X_p
13: X_p transmet R' à x
14: **sinon**
15: Y_p refuse Q
16: **fin si**
17: **sinon**
18: Y_p refuse Q
19: **fin si**

le cas des grilles électriques intelligentes, où l'on souhaite analyser la consommation électrique en temps réel, l'organisation responsable de la grille pourrait n'avoir accès qu'à la consommation globale des immeubles (agrégat d'appartements) et non pas aux consommations individuelles de chaque logement. Sachant que certains agrégats d'espaces peuvent posséder plus de deux niveaux, chaque nouvelle transition des données donne lieu à des agrégations qui réduisent la portée individualisante des informations d'origine : une ville agrège des quartiers, qui agrègent des immeubles, qui agrègent des appartements.

Outre la possibilité d'un contrôle globalisé des données par l'administrateur de l'espace, les points d'échanges présentent un certain intérêt du point de vue de la sécurité. En effet, sachant que les points d'échanges sont amenés à être déployés sur des appareils possédant des capacités matérielles accrues, ceux-ci peuvent chiffrer automatiquement les requêtes inter-espaces. Si l'on considère l'Algorithme 4.1, le chiffrement des requêtes s'effectue au niveau des étapes de communication entre les deux points d'échange (lignes 3 et 13). Pour ce faire, les deux points d'échanges peuvent utiliser leurs couples de clés asymétriques respectifs.

Les espaces agrégés peuvent aussi fournir des méthodes d'anonymisation basée sur les techniques d'*onion routing* [169] ; chaque point d'échange ajoutant une couche de chiffrement aux messages qui les traversent. L'exploitation de cette technique spécifique n'est pas implémentée dans notre prototype et demeure un sujet de recherche futur.

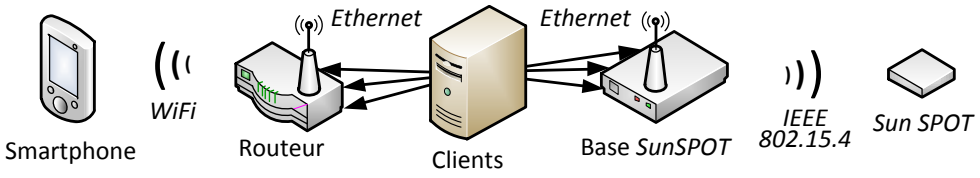


Figure 4.6 – Environnement expérimental.

4.3 Évaluation de *Dioptase*

Les expériences présentées dans cette section ont deux objectifs. Premièrement, nous voulons montrer que la phase de personnalisation permet d'utiliser *Dioptase* sur des appareils de capacités matérielles différentes. Chacun de ces appareils peut alors servir des flux *HTTP* efficacement, conformément aux ressources disponibles. Le choix de cette implémentation de flux pour notre expérience est justifié par le fait que les flux *HTTP* sont les plus verbeux ; les performances au moyen de flux *CoAP* seront forcément meilleures en pratique. Dans un second temps, nous cherchons à mesurer le surcoût induit par l'interpréteur de code *DiSPL* par rapport à l'utilisation d'opérateurs compilés.

Pour ces expériences, nous nous concentrons sur deux objets : un smartphone *Galaxy Nexus* et un capteur *Sun SPOT*. Le *Galaxy Nexus* est un système mobile, fonctionnant sous *Android* 4.3 « Jelly Bean », et possédant des capacités matérielles supérieures (objet intelligent). Il est en effet équipé d'un microprocesseur double-cœur 1.2GHz (*ARM Cortex-A9*) ainsi que d'un gigaoctet de mémoire vive. Les *Sun SPOT* sont des capteurs embarqués sans fil, développés par *Sun Microsystems* (aujourd'hui *Oracle*), et qui embarquent une machine virtuelle *Java* allégée, appelée *Squawk*. La sixième version des *Sun SPOT* est équipée d'un microprocesseur 400 MHz (*AT91SAM9G20*) et d'un mégaoctet de mémoire vive.

À des fins de comparaison, nous déployons sur ces deux appareils une version personnalisée de *Dioptase* (~209 Ko) incluant l'ensemble des modules constituant l'intergiciel, à l'exception des *plugins* de compression et de chiffrement qui ne sont pas utilisés lors de ces expériences.

4.3.1 Évaluation des capacités de diffusion

Notre première expérience analyse la capacité de *Dioptase* à servir des flux de données efficacement en utilisant l'implémentation de flux *HTTP*. À cette fin, un producteur est déployé sur les objets et acquiert des données depuis les capteurs de lumière qui sont disponibles sur les deux appareils. Toutes les 500 millisecondes, le producteur effectue une mesure et la transmet (~100 o/s) à chaque consommateur connecté. Les différents consommateurs sont déployés sur une machine de bureau standard et leur nombre est adapté en fonction de l'appareil ciblé, sachant que le *Sun SPOT* possède des capacités matérielles très inférieures au *Galaxy Nexus*. Toutes les expériences génèrent des mesures

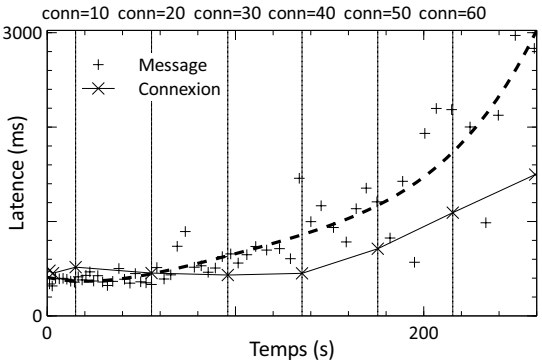


Figure 4.7 – Latence (*Sun SPOT*).

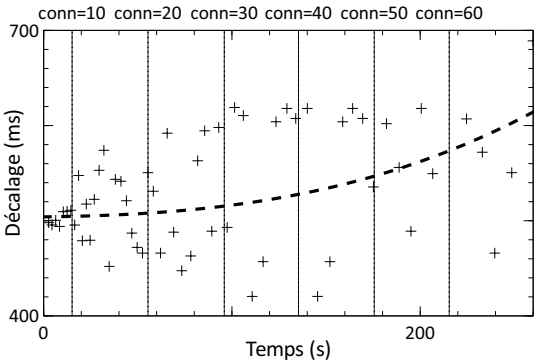


Figure 4.8 – Gigue (*Sun SPOT*).

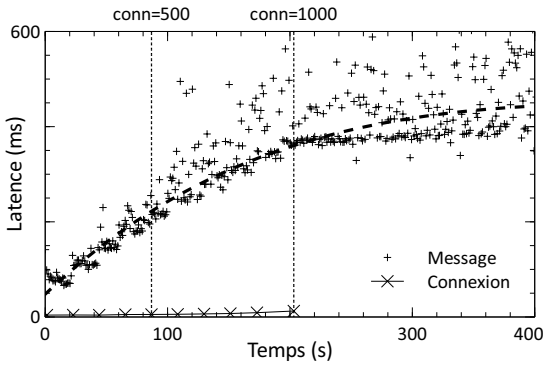


Figure 4.9 – Latence (*Galaxy Nexus*).

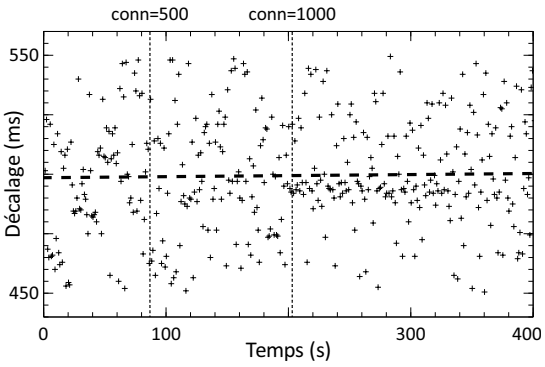


Figure 4.10 – Gigue (*Galaxy Nexus*).

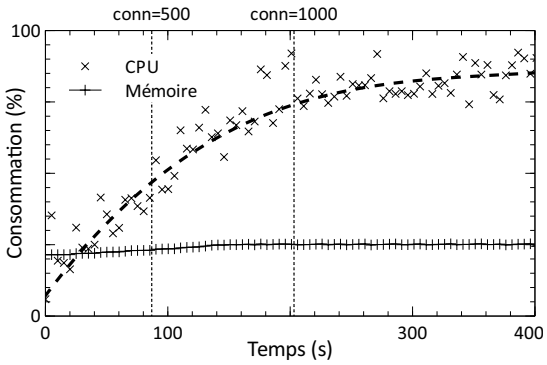


Figure 4.11 – Charge *CPU* et mémoire consommée (*Galaxy Nexus*).

de performance brutes directement dans les espaces de stockage internes aux appareils ; le smartphone et le *Sun SPOT* embarquent tous les deux une mémoire flash dont la taille est, respectivement, de 16 Go et 4 Mo. Avant le démarrage de l'expérience, les appareils sont synchronisés au moyen d'un paquet *UDP* diffusé en mode *broadcast*, le décalage mesuré entre les horloges étant inférieur à 10 millisecondes en pratique. À la fin de l'expérience, les données enregistrées sur les appareils sont collectées pour être traitées.

Comme montré sur la Figure 4.6, la communication entre les clients et les *Sun SPOT* est effectuée au travers d'une base utilisée comme routeur entre le réseau *Ethernet* et le réseau *IEEE 802.15.4*. L'expérience se déroule en deux phases, consistant à accroître le nombre de clients de manière périodique : (i) toutes les deux secondes, le client ouvre une

nouvelle connexion vers le *Sun SPOT* jusqu'à atteindre dix connexions simultanées, puis (ii) toutes les 40 secondes, le client ouvre dix connexions supplémentaires afin de placer l'appareil en situation de stress. Le temps nécessaire à l'établissement des connexions *TCP*, le temps écoulé entre deux messages et les temps de production et de réception d'une mesure, incluant le temps de transmission et le temps de traitement par l'intergiciel, sont collectés. La Figure 4.7 présente la durée moyenne d'ouverture d'une connexion *TCP* et le temps de transmission d'une mesure de luminosité. La Figure 4.8 montre, quant à elle, le temps écoulé entre deux réceptions de message ; dans l'idéal cette durée devrait être constante et rester proche de l'intervalle de production : un élément toutes les 500 millisecondes.

L'expérience sur le smartphone est effectuée cette fois-ci au travers d'une connexion *WiFi 802.11g* en mode point d'accès. Cette expérience collecte les mêmes informations que précédemment. Cependant, le nombre de connexions ouvertes est plus élevé, pour amener le smartphone à une situation de stress malgré ses capacités matérielles supérieures. L'expérience démarre donc avec 100 connexions et, toutes les 20 secondes, 100 consommateurs supplémentaires se connectent au smartphone jusqu'à un total de 1000. Les Figures 4.9 et 4.10 présentent les mêmes résultats que l'expérience avec le *Sun SPOT* : durée d'ouverture de connexion, temps de transmission et gigue. Contrairement aux *Sun SPOT*, le smartphone nous permet d'obtenir des informations sur la consommation de mémoire et la charge *CPU*, grâce aux fichiers systèmes */proc/stat* et */proc/meminfo*. La Figure 4.11 présente ces mesures, acquises toutes les cinq secondes sur l'appareil, cette longue durée étant choisie pour éviter que les mesures s'influencent les unes les autres.

Comme nous pouvons l'imaginer, les ressources disponibles diminuent avec le nombre de flux servis, jusqu'à un seuil critique bien visible sur la Figure 4.7 qui présente la gigue pour le *Sun SPOT*. En effet, après 40 connexions, le *Sun SPOT* est surchargé et la latence croît rapidement (les opérations d'encodage des éléments de flux sont retardées) et, de fait, la gigue croît elle aussi (voir Figure 4.8). En ce qui concerne le smartphone, le système est surchargé au-delà de 1000 connexions et *Android* met fin à l'application pour préserver les ressources nécessaires à son fonctionnement. Les données montrées sur les Figures 4.9 et 4.10 sont donc limitées à ce nombre de connexions, pour lesquelles les ressources consommées restent stables.

Comparer ces performances à des systèmes existants est particulièrement délicat puisque les classes d'objets, les objectifs et les fonctionnalités de *Dioptase* sont uniques à notre connaissance. En effet, *Dioptase* est conçu pour s'exécuter sur de nombreux objets et les abstraire sous la forme de ressources génériques de calcul et de stockage sur lesquelles il est possible de déployer dynamiquement des tâches de traitement continu interprétées à la volée. À l'heure actuelle, les systèmes focalisés sur les réseaux de capteurs sans fil mettent l'accent sur les performances là où *Dioptase* met l'accent sur l'unification, la généricité et la flexibilité. À l'inverse, les systèmes que l'on retrouve au niveau des services tiers pour l'Internet des objets (services Web, *cloud*, *DSMS* génériques, etc.) sont conçus

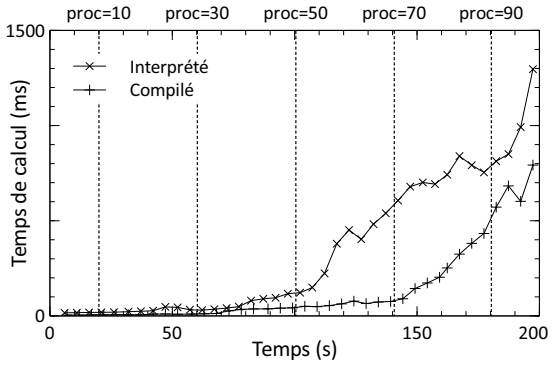


Figure 4.12 – Temps de calcul (*Sun SPOT*).

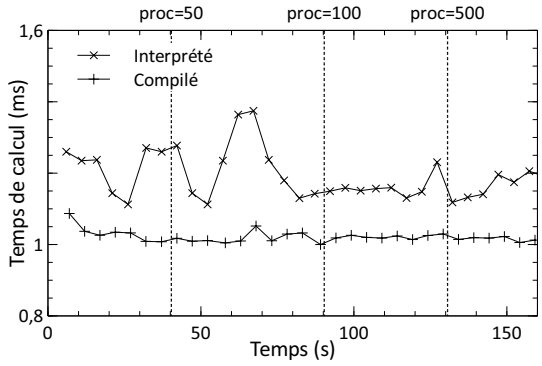


Figure 4.13 – Temps de calcul (*Galaxy Nexus*).

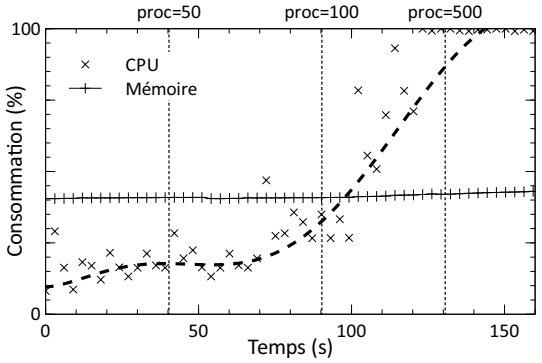


Figure 4.14 – Jointure interprétée (*Galaxy Nexus*).

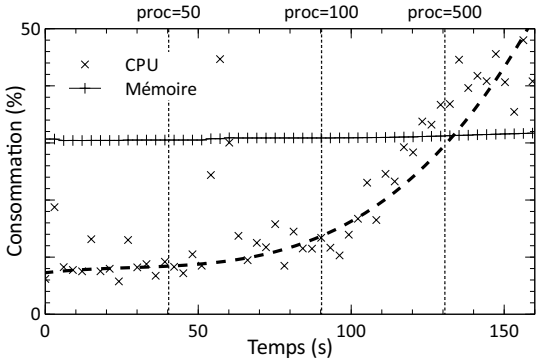


Figure 4.15 – Jointure compilée (*Galaxy Nexus*).

pour s'exécuter sur de puissantes infrastructures de calcul et ne ciblent, au mieux, que les objets les plus performants ; par exemple des smartphones. Cependant, les résultats obtenus peuvent être discutés au regard des scénarios usuels considérés pour l'Internet des objets [7]. En effet, si un objet peut servir environ 30 flux *HTTP*, à une fréquence de deux mesures par seconde, avec une latence réduite (environ 500 ms), alors de nombreux scénarios peuvent être satisfaits. Par exemple, considérons le projet *SmartPark*[1], où les conducteurs sont synchronisés et guidés vers des places de parking libres. Pour ce faire, chaque place est équipée d'un capteur de présence qui transmet des informations à des véhicules équipés d'une interface de communication sans fil. Si chaque emplacement peut servir une trentaine de flux, comme montré dans notre expérience, alors l'ensemble du parking peut gérer et traiter des milliers de flux, ce qui est plus que nécessaire pour ce scénario.

4.3.2 Évaluation des capacités de traitement

Notre seconde expérience porte sur la capacité de *Dioptase* à supporter le déploiement et l'exécution dynamique de tâches et consiste à évaluer la consommation des ressources

1. *http://smartpark.epfl.ch* (accédé le 08/12/2014).

pour des transformateurs exécutant des traitements compilés et interprétés. Le traitement choisi est ici une implémentation de la jointure interne basée sur des tables de hachage [170], que nous exécutons plusieurs fois en parallèle sur deux flux de mesures de luminosité, l'un produit localement par l'objet sur lequel nous effectuons les mesures et l'autre produit par un second objet (*Sun SPOT*). Tout comme pour notre expérience de diffusion de flux, les producteurs lisent les capteurs de luminosité toutes les 500 millisecondes. L'implémentation considérée pour la jointure possède un état interne qui croît proportionnellement aux nombres de valeurs uniques dans chaque flux d'entrée ; une table de hachage étant maintenue pour chacun de ces flux. Lorsqu'un nouvel élément x est lu depuis une entrée y, l'opérateur vérifie s'il existe dans toutes les autres tables. Le cas échéant, l'élément est inscrit dans le flux de sortie et est ajouté à la table y si nécessaire.

L'expérience sur *Sun SPOT* se déroule en deux étapes, aussi bien pour la jointure compilée que la jointure interprétée : (i) toutes les cinq secondes un nouveau transformateur est déployé jusqu'à atteindre cinq transformateurs parallèles, puis (ii) 10 nouveaux transformateurs sont déployés toutes les 40 secondes jusqu'à atteindre le point de surcharge de l'appareil. Comme nous l'avons déjà mentionné, nous ne pouvons pas obtenir d'informations sur les ressources consommées sur les *Sun SPOT* et, de fait, nous ne mesurons que le temps passé par chaque transformateur pour exécuter la jointure à la réception d'un élément en entrée. Le temps de calcul est, en effet, une image de la consommation de mémoire et de la charge *CPU*, ce temps ayant tendance à croître lorsque l'appareil est surchargé. La Figure 4.12 montre le temps d'exécution moyen au fur et à mesure de l'ajout des transformateurs, le point de surcharge étant atteint avec une centaine de transformateurs.

L'expérience sur smartphone est similaire et comporte elle aussi deux étapes : l'appareil démarre avec 10 transformateurs et, toutes les 10 secondes, déploie 10 transformateurs supplémentaires. Lorsque le smartphone atteint 100 transformateurs, le nombre de transformateurs déployés toutes les 10 secondes passe à 100 pour atteindre plus rapidement la surcharge. Cette fois-ci, en plus du temps d'exécution présenté sur la Figure 4.13, nous pouvons obtenir des informations à propos de la mémoire consommée et de la charge *CPU*, comme montré sur les Figures 4.14 (jointure interprétée) et 4.15 (jointure compilée).

Comme pressenti, les jointures interprétées sont plus coûteuses que leurs homologues compilés, du fait de l'exécution de l'arbre de syntaxe abstraite (*AST*) et de son maintien en mémoire. Les figures montrent globalement que les jointures interprétées ont un temps d'exécution deux fois plus élevé que les jointures compilées. Cette différence est beaucoup moins significative sur le smartphone, qui possède beaucoup de mémoire pour stocker les *AST*. La différence est inférieure à 40 microsecondes dans le pire des cas, certains pics de consommation étant aussi une conséquence du ramasse-miette exécuté par la machine virtuelle *Java*. Sur le *Sun SPOT*, la surcharge survient vers 60 jointures interprétées et 90 jointures compilées. Comme nous l'avons évoqué, cette surcharge n'est pas due au *CPU*,

qui est largement surdimensionné pour ce type de traitement, mais de la mémoire qui se trouve saturée par la conservation des *AST*.

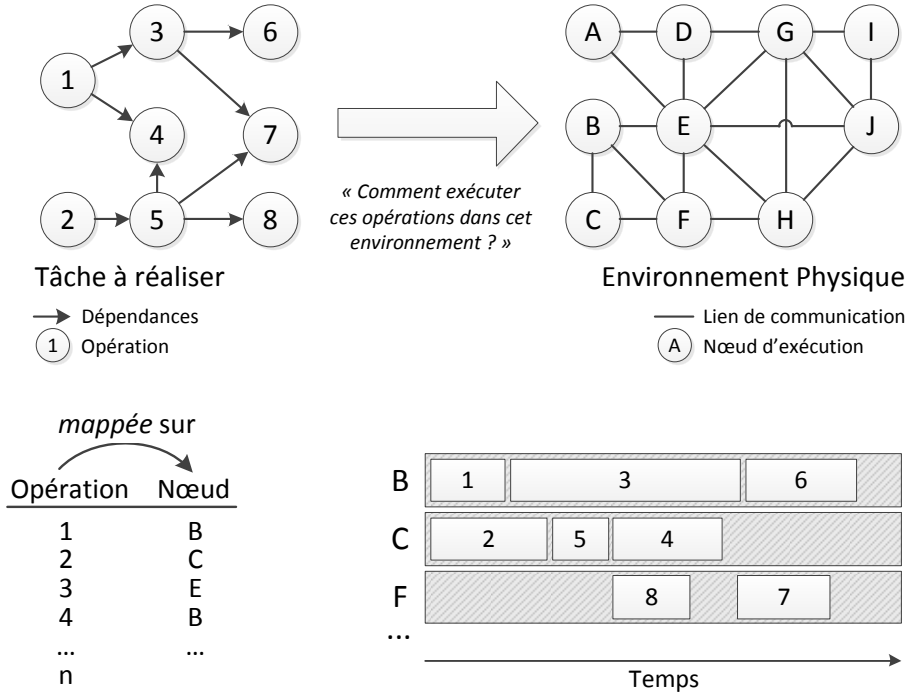
Comme précédemment, les résultats obtenus sont satisfaisants dans l'absolu, mais diffiles à comparer avec des systèmes existants étant donné que, à notre connaissance, les *DSMS* pour systèmes embarqués ne gèrent pas le déploiement et l'exécution dynamique de tâches complètes. En outre, cette implémentation de la jointure interne est coûteuse, du fait de son coût en *CPU* et en mémoire croissant au cours du temps contrairement à des traitements tels que le comptage, la projection ou la sélection, qui peuvent être calculés en temps et en espace constant. Toujours en considérant les scénarios Internet des objets usuels, la capacité à exécuter en parallèle plus de 60 traitements complexes pouvant être spécifiés à tout moment (respectivement plus de 90 traitements compilés prédéfinis à l'avance) sur un seul objet est compatible avec les besoins de l'Internet des objets.

Chapitre 5

Déploiement automatique de services dans l'Internet des objets

---

Dans le cadre de l'Internet des objets, une application orientée flux doit pouvoir être déployée automatiquement dans un réseau d'objet hétérogène, ce qui implique de déterminer sur quels appareils les différents services de cette application doivent être placés. Cette problématique a été étudiée dans de nombreux autres contextes sous le nom du *problème de* mapping *de tâche*. Nous présentons *TGCA*, une nouvelle formulation de ce problème pour l'Internet des objets et une méthode de résolution correspondante.



(a) *Mapping* de tâche (b) Ordonnancement de tâche (*task scheduling*)

Figure 5.1 – Le problème général de *mapping* et d'ordonnancement de tâche.

Les utilisateurs sont amenés à interagir avec l'Internet des objets au travers d'un écosystème complexe d'applications et de services consommant toutes sortes de flux de données issus de sources multiples. La construction de ces applications comme assemblage composite de flux (*tâches*) nécessite de fournir aux développeurs des moyens sophistiqués pour exprimer les traitements continus et les déployer automatiquement dans les réseaux d'objets. Pour ce faire, nous avons introduit au Chapitre 3 la notion de graphe logique pour décrire des tâches qui composent des services continus et manipulent des flux de données. Puis, nous avons introduit au Chapitre 4 l'intergiciel *Dioptase* pour disposer d'une interface unifiée de déploiement de services continus. Ce chapitre traite donc de l'étape suivante qui consiste à déterminer comment les services continus qui constituent un graphe logique peuvent être déployés dans un réseau d'appareils communicants, ou nœuds [1].

La recherche d'une configuration de déploiement optimale, c'est-à-dire qui minimise ou maximise certains paramètres au regard des caractéristiques des tâches et des nœuds, s'appelle le « problème de *mapping* de tâche » (*task mapping problem*). En pratique, il est souvent associé à une problématique sœur, l'ordonnancement de tâche (*task scheduling*), qui consiste à établir dans quel ordre les traitements doivent être exécutés au sein des différents nœuds, de façon à maximiser ou minimiser certains paramètres (typiquement,

1. Dans le Chapitre 2, nous utilisions le terme « nœud » pour désigner un appareil restreint en ressources équipé de capteurs et d'actionneurs. Cependant, tout au long de ce chapitre, ce terme sera utilisé dans son sens plus général : un appareil quelconque possédant des capacités de calcul.

minimiser le temps d'exécution total) [171]. Cependant, dans le cadre du traitement de flux, la durée d'exécution des traitements n'est pas connue à l'avance et s'établit habituellement sur de longues durées. Il n'est donc pas pertinent de considérer l'ordre d'exécution au sein d'un même calculateur, seul le parallélisme devant éventuellement être pris en compte (*mapping* de tâche partagé).

Conformément à ce que nous avons vu au Chapitre 2, le problème de *mapping* de tâche comporte deux étapes :

— le *calcul d'une configuration initiale* (ou *mapping initial*), à partir des informations connues avant le déploiement ;

— l'*adaptation dynamique*, qui consiste à faire évoluer dynamiquement la configuration initiale une fois déployée, en fonction des évènements qui peuvent survenir (variations du débit, apparition-disparition de nœuds, etc.).

En fonction des approches, les deux phases ne sont pas traitées avec la même importance. On parlera de *mapping statique* si l'accent est mis sur la recherche d'une configuration optimale qui changera peu au cours du temps ; les changements nécessitant généralement de recalculer complètement le *mapping*. Au contraire, on parlera de *mapping dynamique* si l'accent est mis sur les mécanismes de migration et de duplication qui font converger le réseau vers une solution. Enfin, de rares approches, que nous appellerons *mapping hybride*, proposent de traiter les deux aspects à parts égales, notamment en recherchant une configuration initiale résistante aux perturbations et qui, de fait, assure une convergence plus rapide lors d'un changement des caractéristiques des nœuds, des flux ou du réseau [137].

Étant donnée la nature très générale du problème de *mapping* de tâche, qui consiste à établir comment associer des traitements à des nœuds de façon à minimiser le coût global, celui-ci a été étudié dans de nombreux contextes : le *calcul parallèle*, où un ensemble de processus ou d'instructions doit être déployé dans une architecture multiprocesseur ou multicœur [171], la *co-conception matérielle-logicielle* (*hardware/software codesign*), où une spécification fonctionnelle doit être partitionnée en spécifications matérielles et logicielles [2] [172], et, bien entendu, les *systèmes distribués* où un ensemble de traitements doit être déployé dans un réseau, une grille, un cluster ou le *cloud* [173].

Étant un cas particulier de systèmes distribués, il est naturel que les réseaux de capteurs et d'actionneurs sans fil soient un sujet d'étude pour le problème de *mapping* de tâche, avec comme spécificité la minimisation de la consommation énergétique [174]. Comme nous l'avons déjà mentionné, l'Internet des objets hérite en partie des problématiques propres aux réseaux de capteurs, mais introduit toutefois des caractéristiques propres du fait de ses nombreux cas d'utilisation. Par exemple, le problème du *shared sensing*, où les utilisateurs ouvrent leurs capteurs à des requêtes extérieures, nécessite de répartir équitablement la

2. Dans le cadre de la conception de circuits intégrés complexes, il s'agit concrètement de déterminer quelles fonctions doivent être implémentées physiquement au niveau matériel et quelles fonctions doivent être implémentées au niveau logiciel, en tenant compte de leurs interdépendances.

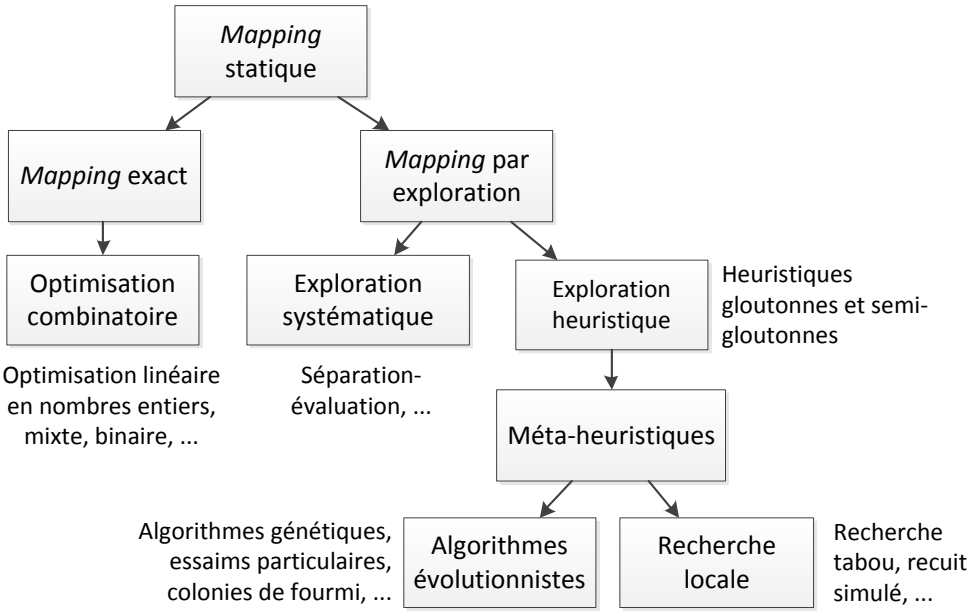


Figure 5.2 – Différentes approches de *mapping* statique.

charge (*load balancing*). En effet, puisque les objets sont mis à disposition gracieusement, il paraît fondamental d'éviter que seuls certains d'entre eux soient constamment surchargés ou vidés de leur énergie, du fait de propriétés avantageuses conduisant le système de calcul des configurations à les sélectionner systématiquement.

5.1 Problème dit de *mapping de tâche*

Globalement, le calcul d'une configuration initiale est un problème d'optimisation considéré comme très difficile du point de vue algorithmique, ce même dans ses formulations les plus simples [175]. En pratique, la recherche de cette configuration nécessite d'employer des méthodes d'approximation ou des heuristiques [3], conçues spécifiquement pour tenir compte des particularités du domaine considéré : l'ensemble des caractéristiques (réseau, tâches, nœuds), le ou les objectifs à minimiser ou maximiser, et les fonctions d'évaluation desdits objectifs.

Du fait de la complexité du problème, on retrouve de nombreuses approches destinées à le résoudre de manière exacte ou approchée, comme le montre la Figure 5.2 [176]. Les méthodes de résolution exacte se basent sur des problèmes d'optimisation combinatoire dont on sait déterminer mathématiquement la solution optimale ; par exemple, l'optimisation linéaire et ses spécialisations [177]. Cependant, ces méthodes de résolution exacte peuvent être très complexes en pratique, c'est-à-dire nécessiter beaucoup de temps et de ressources de calcul. Les méthodes exploratoires permettent de trouver des solutions optimales ou sous-optimales en explorant « intelligemment » l'espace des solutions possibles,

3. Une heuristique est une méthode pour calculer des solutions sous-optimales, typiquement pour des problèmes difficiles à résoudre de manière exacte.

soit de manière systématique (recherche exhaustive) soit en exploitant des informations la structure des « bonnes » solutions : heuristiques et métaheuristiques [4]. Enfin, la classification présentée sur la Figure 5.2 n'est pas exhaustive, d'autres méthodes (exactes ou approchées) ont été utilisées par les chercheurs pour résoudre le problème de *mapping* de tâche : théorie des jeux [179], programmation dynamique [180], programmation par contrainte [181], etc.

5.1.1 Formulation historique du problème de *mapping* de tâche

L'une des plus anciennes formulations du problème de *mapping* [175] pour un système réel à été réalisée dans le cadre du calcul parallèle au sein d'une *machine à éléments finis* (*finite element machine*, ou *FEM*), c'est-à-dire une matrice de calcul multiprocesseur où chaque processeur est (i) physiquement relié à ses huit plus proches voisins (communication rapide) et (ii) capable d'échanger avec n'importe quel autre processeur au travers d'un bus de communication global (communication lente). Ici, le problème de *mapping* consiste à placer les traitements communicants sur des processeurs directement connectés, pour maximiser la vitesse de calcul [175].

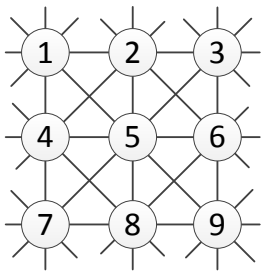


Figure 5.3 – *FEM* 3×3

Dans cette approche, les tâches à déployer sont décrites sous la forme d'un graphe simple $G_p = (V_p, E_p)$ où les sommets sont des tâches et les arêtes des liens de communication. L'architecture d'exécution est décrite de la même façon, $G_a = (V_a, E_a)$, où les sommets sont des processeurs et les arêtes des connexions directes entre ces derniers. En prenant pour postulat initial que $|V_p| = |V_a|$, par exemple en ajoutant des tâches vides lorsque $|V_p| < |V_a|$, une configuration est alors définie comme une bijection $m : V_p \to V_a$ qui à une tâche associe un processeur. La recherche de la configuration idéale consiste à maximiser la valeur d'une solution m, notée $|m|$, estimée comme le nombre de liens entre les traitements qui sont effectivement associés aux liens entre les processeurs :

$$|m| = \frac{1}{2} \sum_{(x,y) \in V_p^2} g_p(x,y) \times g_a(m(x), m(y))$$

$$\text{avec } g_p : V_p^2 \to \{0,1\} \text{ et, respectivement, } g_a : V_a^2 \to \{0,1\}$$

$$g_p(x,y), \text{ resp. } g_a(x,y) \ = \begin{cases} 1 & \text{si } (x,y) \in E_p, \text{ resp. } E_a, \\ 0 & \text{sinon.} \end{cases}$$

De là, deux questions intéressantes se posent quant à la recherche d'une solution au problème :

4. Une métaheuristique est une stratégie générale d'exploration d'un espace de solution quelconque, souvent construite comme une métaphore d'un comportement physique ou naturel (p. ex. la métallurgie pour le recuit simulé ou la théorie de l'évolution pour les algorithmes génétiques) [178].

(a) Étant donné un G_p et un G_a, existe-t-il une configuration m idéale qui satisfasse $|m| = |V_p|^2 = |V_a|^2$?

(b) S'il n'existe pas de m idéal, comment déterminer la configuration dont la valeur $|m|$ est maximale ?

Répondre à la première question revient à trouver un isomorphisme entre les graphes V_p et V_a [175]. Pour rappel, deux graphes $G_1 = (V_1, E_1, \varphi_1)$ et $G_2 = (V_2, E_2, \varphi_2)$ sont dits *isomorphes*, noté $G_1 \simeq G_2$, s'il existe une bijection (ou isomorphisme) entre les sommets de G_1 et G_2 qui préserve les arêtes. Formellement, étant donné la bijection $f : V_1 \to V_2$, on a : $\forall e_1 \in E_1, \varphi_1(e_1) = \{x, y\} \Leftrightarrow \exists e_2 \in E_2$ tel que $\varphi_2(e_2) = \{f(x), f(y)\}$. Le problème de l'isomorphisme de graphe, qui consiste à déterminer si G_1 et G_2 sont isomorphes en recherchant un isomorphisme f, est connu pour être *NP*-difficile et le meilleur algorithme de résolution a un temps d'exécution de $2^{O(\sqrt{n \log n})}$, avec n le nombre de sommets [182]. Concrètement, s'il était possible de répondre à la question (a) avec un algorithme en temps polynomial, alors le même algorithme pourrait être utilisé pour résoudre le problème d'isomorphisme de graphe et inversement [175].

Concernant la seconde question, il s'agit d'un problème similaire à celui de l'*affectation quadratique* (*quadratic assignment problem*, ou *QAP*) [183], où étant donné n objets et n zones, on cherche à placer les objets possédant le plus d'affinité (p. ex. utilisés en commun) dans les zones les plus proches. Formellement, on pose deux matrices C et D de taille $n \times n$, où C_{ij} exprime l'affinité entre les objets i et j, et où D_{kl} exprime la distance entre les zones k et l. On cherche alors à trouver une bijection a (appelée *affectation*) qui associe un objet à une zone, de façon à minimiser $\sum_{i,j} C_{ij} D_{a(i)a(j)}$. Concrètement, le *QAP* se réduit au problème de mapping si les matrices C et D sont réduites à des valeurs binaires [175]. Ce problème est *NP*-difficile et ne possède pas d'algorithme de résolution exacte en temps polynomial à l'heure actuelle [184].

Aussi, l'algorithme proposé dans [175] pour résoudre le problème de *mapping* de tâche est un algorithme heuristique semi-glouton [5], qui (i) recherche les couples traitement-processeur qui maximisent la valeur de la configuration m et (ii) effectue des permutations aléatoires dans l'espace des solutions pour éviter de bloquer l'algorithme dans des solutions inintéressantes du point de vue global. Le détail de ce fonctionnement est présenté dans l'Algorithme 5.1, où la bijection m est représentée sous la forme d'une matrice M de taille $|V_p| \times |V_a|$ (carrée ici, $|V_p| = |V_a|$), avec $M_{ij} = 1$ si le traitement i est associé au processeur j, 0 sinon.

La complexité temporelle de l'algorithme est de $O(|V_a|^2)$ pour la partie effectuant les permutations. En ce qui concerne la boucle générale, la valeur de $|m|$ ne peut pas excéder

5. Un algorithme glouton désigne généralement un algorithme d'optimisation qui effectue, étape par étape, des choix localement optimaux, et qui converge peu à peu vers un optimum global. Lorsque le résultat est un sous-optimum global, on parle alors d'heuristique gloutonne, et lorsque les décisions de ces heuristiques sont en partie basées sur de l'aléatoire, on parle d'heuristique semi-gloutonne [185].

Algorithme 5.1 Construction d'une configuration de déploiement pour une *FEM*

Entrées : G_p, G_a (défini par la taille n de la *FEM* $n \times n$)

remplir M aléatoirement*
$best \leftarrow M$, $done \leftarrow$ **faux**, $continue \leftarrow$ **faux**
tant que $done =$ **faux faire**
  répéter
    $continue \leftarrow$ **faux**
    pour tout sommet $x \in V_p$ **faire**
      1. évaluer $|m|$ pour chaque association de x avec un autre sommet $y \in V_a$
      2. sélectionner la permutation (x, y) permettant de réaliser le meilleur gain α
      3. si $\alpha \geq 0$, valider la permutation
      4. si $\alpha > 0$, $continue \leftarrow$ **vrai**
    fin pour
  jusqu'à $continue =$ **faux**
  si $|m| < |best|$ **alors**
    $done \leftarrow$ **vrai**
  sinon
    $best \leftarrow M$
    modifier aléatoirement* n associations de M
  fin si
fin tant que
retourner $best$

* en respectant l'isomorphisme m : un seul traitement associé à un seul processeur et inversement.

$4 \times |V_a|$ (8 liens par processeur) et le gain minimal entre deux itérations est de 1. Ainsi, dans le pire des cas, la complexité de l'algorithme complet est de $O(|V_a|^2 \times 4|V_a|) = O(|V_a|^3)$.

5.1.2 Optimisation linéaire

Un problème d'*optimisation linéaire* (*linear programming*, ou *LP*) [186] est un problème d'optimisation dans lequel la fonction objectif à optimiser est une fonction affine et où les contraintes sont exprimées sous la forme d'équations ou d'inéquations linéaires dont les inconnues sont les variables de décision du problème. Soit un problème d'optimisation linéaire à n variables x_i et m contraintes, alors la formulation générale est la suivante :

$$\textbf{maximiser (ou minimiser)} \quad \sum_{i=1}^{n} c_i x_i$$

$$\textbf{tel que} \quad \sum_{i=1}^{n} a_{1i} x_i \diamond b_i$$

$$\vdots$$

$$\sum_{i=1}^{n} a_{mi} x_i \diamond b_m$$

avec $\diamond$ l'un des opérateurs suivants : $\leq, \neq, =$.

Pour un problème d'optimisation linéaire à n variables, l'espace des solutions est un espace à n dimensions dans lequel les contraintes forment un ensemble d'hyperplans qui délimitent la région de l'espace où se trouve l'ensemble des solutions réalisables (*feasible region*). À partir du moment où les variables sont restreintes à des valeurs réelles non négatives, alors la solution optimale se situe à l'une des extrémités de cette région et peut être trouvée en un temps raisonnable même pour des problèmes contenant plusieurs milliers de variables [186].

L'*optimisation linéaire en nombres entiers* (*integer linear programming*, ou *ILP*) [187] est une variation du problème d'optimisation linéaire, qui a été régulièrement utilisée, entre autres, pour la résolution du problème de *mapping* de tâche [174, 176]. Les problèmes d'*ILP* introduisent de nouvelles contraintes, en restreignant les variables de décision à des valeurs entières ; lorsque seules certaines variables sont sujettes à cette contrainte, on parle d'*optimisation linéaire mixte en nombres entiers* (*mixed integer linear programming*, ou *MILP*). Cette restriction, en apparence simple, conduit à un accroissement significatif de la complexité, résoudre un problème d'*ILP* étant *NP*-difficile tandis que la résolution d'un problème de *LP* ne l'est pas [187]. La résolution de tels problèmes peut donc être très difficile en pratique.

Un exemple d'utilisation de *MILP* pour la formulation d'un problème de *mapping* de tâche pour les réseaux de capteurs sans fil est celui présenté dans [174], où l'on cherche à réduire les coûts énergétiques liés à l'exécution, la communication et au routage. Ici, le comportement du réseau de capteurs est considéré cyclique, le réseau revenant à son état initial à la fin d'un cycle. Concrètement, un cycle correspond à plusieurs exécutions de chaque tâche sur les nœuds, deux tâches communicantes i et j étant caractérisées par le nombre de bits de données s_{ij} qu'elles échangent par invocation. Pour pouvoir rechercher une configuration de déploiement de m tâches sur n nœuds, cette approche représente les coûts énergétiques sous la forme de deux matrices : (i) la matrice d'exécution $\mathcal{T}$ de taille $m \times n$ où chaque $\mathcal{T}_{ik}$ représente l'énergie consommée par une exécution de la tâche i sur le nœud k, et (ii) la matrice de routage $\mathcal{R}$ de routage de taille $n \times n \times n$ où chaque $\mathcal{R}_{\beta\gamma k}$ représente l'énergie consommée par le nœud k pour router un bit de donnée du nœud β vers le nœud γ. La consommation énergétique sur un nœud k, pour le calcul

C^k_{comp} et pour la communication C^k_{comm}, est calculée en agrégeant les différents coûts liés aux f_i exécutions d'une tâche pendant le cycle. Si l'on note DE l'ensemble des paires (i, j) de tâches communicantes et $x_{ik} = 1$ lorsque la tâche i est déployée sur un nœud k et 0 sinon, on a :

$$C^k_{comp} = \sum_{i=1}^{m} x_{ik} \cdot f_i \cdot \mathcal{T}_{ik}$$

$$C^k_{comm} = \sum_{(i,j)\in DE} \sum_{\beta=1}^{n} \sum_{\gamma=1}^{n} f_i \cdot s_{ij} \cdot x_{i\beta} \cdot x_{j\gamma} \cdot \mathcal{R}_{\beta\gamma k}$$

Deux problèmes sont alors étudiés, le premier consistant à minimiser la consommation énergétique individuelle de chaque nœud et le second consistant à minimiser la consommation totale du réseau. Par exemple, le problème de minimisation de la consommation globale est le suivant :

$$\begin{aligned} \textbf{minimiser} \quad & \sum_{k=1}^{n} C^k_{comp} + C^k_{comm} \\ \textbf{tel que} \quad & \sum_{k=1}^{n} x_{ik} = 1 \quad \forall i \in [\![1, m]\!] \\ & C^k_{comp} + C^k_{comm} \leq e^0_k \quad \forall k \in [\![1, n]\!] \\ & \text{avec } e^0_k \text{ l'énergie disponible sur le nœud } k. \end{aligned}$$

Une méthode de résolution heuristique est alors proposée sous la forme d'un algorithme glouton qui (i) trie les tâches communicantes (i, j) par ordre décroissant de volume de données échangé ($f_i \cdot s_{ij}$) puis (ii) itère sur l'ensemble trié et associe les tâches i et j à des nœuds k et l de façon à minimiser le coût énergétique total, ce qui revient à minimiser le coût individuel de calcul de i et j sur k et l et à rechercher les k et les l entre lesquels existe le plus court chemin possible.

5.1.3 Métaheuristiques

Pour rechercher des solutions approchées au problème de *mapping* de tâche, de nombreuses métaheuristiques ont été utilisées dans la littérature.

La *recherche tabou* (*tabu search*) [188] est une métaheuristique qui permet d'explorer le voisinage $N(x)$ d'une solution x dans un espace de solutions X. Chaque solution voisine $x' \in N(x)$ est calculée en effectuant un *saut* à partir de x, la notion de saut étant spécifique à l'application considérée. Par exemple, dans le cas du problème de *mapping* de tâche, un saut peut être défini comme une permutation de deux traitements [189] ou le déplacement d'un traitement dans un autre appareil. La recherche tabou consiste alors à partir d'une solution initiale x, par exemple générée aléatoirement, puis à sélectionner la meilleure solution dans le voisinage de x en utilisation une fonction d'évaluation spécifique. Pour éviter de bloquer l'algorithme dans un minimum local ou dans une

oscillation entre deux solutions équivalentes, la recherche tabou fait appel à une mémoire, ou *liste tabou*, dans laquelle sont enregistrées les solutions déjà parcourues, de façon à ce que l'algorithme de recherche ne revienne jamais sur des solutions déjà explorées. Les critères d'ajout des solutions à la liste tabou peuvent dépendre des applications, notamment si l'on dispose d'informations sur la structure des « bonnes » solutions, c'est-à-dire une région particulière de l'espace des solutions à explorer en priorité [188]. Comme tous les algorithmes d'exploration, il est nécessaire de déterminer une condition d'arrêt, celle-ci pouvant être généraliste ou spécifique à l'application. Par exemple, une condition d'arrêt généraliste est celle consistant à arrêter l'algorithme lorsqu'aucune meilleure solution n'est trouvée après un nombre fixé de sauts. Étant donné le grand nombre de paramètres à fournir (fonction d'évaluation, construction d'un voisinage, condition d'arrêt, règles tabou, etc.), une bonne connaissance du problème est requise, des choix mal avisés pouvant conduire à des solutions de mauvaise qualité [190].

Les méthodes évolutionnistes ont, de la même façon, été utilisées pour rechercher des solutions aux problèmes de *mapping* et d'ordonnancement de tâche [176]. Parmi les plus connus, on trouve les *algorithmes génétiques* [191], une méthode d'exploration de l'espace des solutions basée sur la métaphore naturelle de mutation-sélection des populations d'êtres vivants. Ici, une solution est décrite sous la forme d'un « chromosome » encodant les caractéristiques de la solution ; par exemple, le i-ème *gène* du chromosome peut représenter le i-ème traitement, la valeur du gène, ou *allèle* correspondant à l'appareil auquel ce traitement est affecté [192]. La procédure d'exploration consiste alors à générer aléatoirement une population initiale et à évaluer chaque chromosome au moyen d'une fonction spécifique (*fitness function*). À partir des scores obtenus par chaque chromosome, seuls les meilleurs sont sélectionnés dans la population, par exemple en ne conservant que les k chromosomes ayant eu les plus mauvais scores, ou en augmentant les chances d'un chromosome d'être sélectionné proportionnellement à son score (méthode de la *roulette russe*) [193]. Les chromosomes restants sont alors soumis à des transformations génétiques : les *recombinaisons génétiques* (*crossover*), qui consistent à construire de nouveaux chromosomes à partir des gènes de deux chromosomes parents, et les *mutations génétiques*, qui modifient aléatoirement les gènes de certains chromosomes. Enfin, la nouvelle population obtenue est soumise à nouveau au processus de sélection-recombinaison-mutation jusqu'à ce que la condition d'arrêt soit satisfaite. Tout comme dans le cas de la recherche tabou, une condition d'arrêt classique consiste à stopper le processus de recherche lorsqu'aucune meilleure solution n'a été trouvée après un nombre fixé de tours. En outre, les algorithmes génétiques doivent eux aussi être paramétrés finement pour obtenir rapidement des solutions de qualité satisfaisante. Pour ce faire de nombreux opérateurs de mutation et de recombinaison ont été proposés, de façon à garantir une exploration de l'espace des solutions plus ou moins efficace [191].

Qui plus est, d'autres métaheuristiques ont été utilisées pour résoudre le problème de *mapping* de tâche : essaims particulaires [194], colonies de fourmis [195], etc.

$GL = (VL, EL)$	Graphe logique décrivant une tâche.
vl_i	Service continu i.
d^i	Durée d'exécution de vl_i.
IS^i, OS^i	Ensemble des flux d'entrée et de sortie de vl_i.
S_{ij}	Flux circulant de vl_i vers vl_j.
η^i	Facteur de réduction de vl_i.
$omem^i(t)$, $odsk^i(t)$	Fonction de consommation d'espace mémoire et d'espace persistant de vl_i.
$ocpu^i(t)$	Fonction de charge *CPU* de vl_i.
$opwr^i_{run}(t)$, $opwr^i_{com}(t)$	Fonctions de consommation énergétique, resp. pour l'exécution et la communication de vl_i.
C^i	Ensemble des contraintes sur vl_i.
N	Ensemble des objets.
n_j	Objet j.
$nmem^j$, $ndsk^j$	Quantité de mémoire volatile et persistante sur n_j.
$ncpu^j$	Nombre d'instructions pouvant être exécutées par seconde sur n_j.
$npwr^j$	Quantité d'énergie disponible sur n_j.
α_{pwr}	Énergie déjà consommée au démarrage de n_j.
α_{cpu}	Charge *CPU* au démarrage de n_j.
α_{mem}, α_{dsk}	Quantités de mémoire et d'espace persistant déjà consommées au démarrage de n_j.
β_{pwr}	Estimation de la consommation énergétique de n_j au repos.
l^j	Position de n_j.
A	Zone géographique.
t_s	Moment où une tâche est déployée.
δ_l	Durée de vie minimale d'une tâche.
Δ_l	Durée de vie maximale d'une configuration.
M	Configuration de déploiement.
$npwr^j_0(t)$	Énergie totale consommée sur n_j.
$ncpu^j_0(t)$	Charge *CPU* totale de n_j.
$nmem^j_0(t)$, $ndsk^j_0(t)$	Mémoire totale et espace persistant total consommés sur n_j.
$pwr^i(t)$	Énergie totale consommée par le réseau.
$cpu^i(t)$	Charge *CPU* totale consommée par le réseau.
$mem^i(t)$, $dsk^i(t)$	Mémoire totale et espace persistant total consommés par le réseau.
$g_{pwr}(t)$	Fonction objectif pour la consommation énergétique.
$g_{cpu}(t)$	Fonction objectif pour la charge *CPU*.
$g_{mem}(t)$, $g_{dsk}(t)$	Fonction objectif pour la consommation de mémoire et d'espace persistant.
$G(t)$	Fonction objectif agrégée.
ε^{ij}	Erreur potentielle.

Table 5.1 – Récapitulatif des notations utilisées pour *TGCA*.

5.2 *TGCA* : une approche de *mapping* statique pour l'Internet des objets

Au-delà des problématiques d'énergie, de ressources et de qualité de service que l'on retrouve dans les réseaux de capteurs, l'Internet des objets possède des caractéristiques uniques vis-à-vis des précédents environnements distribués dans lesquels le problème de *mapping* de tâche à été étudié. De fait, il est nécessaire d'exprimer une nouvelle formulation du problème, adaptée aux contraintes et aux propriétés de l'Internet des objets. Une telle formulation doit tenir compte de plusieurs aspects :

- Les spécificités matérielles et logicielles des objets : mémoire, capacités *CPU*, énergie, capteurs et actionneurs, bibliothèques logicielles, composants matériels spécifiques (p. ex. chiffrement matériel) et interfaces de communication.
- La nature continue des traitements (services continus) et des données (flux).
- La possibilité de répartir équitablement la charge à travers les objets disponibles, notamment pour des scénarios de *shared sensing*. En effet, si certains objets sont systématiquement surchargés, du fait de leurs propriétés avantageuses, alors ces objets seraient inutilisables pour leurs possesseurs.
- La prise en compte de la concurrence des traitements sur les objets multitâches.
- La qualité de service assurée par les différents objets, en matière d'efficacité, de sécurité et de fiabilité.

À l'exception de la nature continue des données et des calculs, ces différentes caractéristiques ont été étudiées indépendamment, d'une manière ou d'une autre, dans la littérature relative au problème de *mapping* de tâche. Toutefois, il est nécessaire aujourd'hui de les unifier au sein d'une formulation cohérente du problème, capable de représenter les tâches, les objets, les contraintes et les objectifs propres à l'Internet des objets. À cette fin, nous introduisons *Task Graph to Concrete Actions* (*TGCA*), une approche de résolution du problème de *mapping* de tâche pour l'Internet des objets. *TGCA* comprend notamment :

- un modèle de description des coûts liés à l'acheminement et au traitement des flux de données dans des réseaux d'objets hétérogènes ;
- une formulation du problème de *mapping* de tâche spécifique à l'Internet des objets ;
- un algorithme de résolution heuristique permettant de déterminer rapidement des configurations de déploiement sous-optimales de qualité satisfaisante.

Nous avons fait le choix, pour *TGCA*, de privilégier une approche plutôt centrée sur le calcul d'une configuration initiale (*mapping* statique). Lorsque des changements surviennent dans le réseau, il est possible de recalculer les configurations de déploiement et de relancer l'exécution du graphe logique. En effet, notre algorithme de résolution est suffisamment simple pour être implémenté sur certains objets du réseau qui possèderaient des ressources matérielles supérieures ; par exemple, des smartphones.

5.2.1 Formalisation du problème de *mapping* de tâche

Le problème de *mapping* de tâche dans l'Internet des objets consiste à déterminer comment déployer une tâche impliquant des producteurs, des transformateurs, des stockages et des consommateurs dans un réseau d'objets qui communiquent entre eux. Concrètement, les objets considérés ici sont ceux qui disposent des capacités suffisantes pour assurer l'un de ces quatre rôles. À l'inverse, les objets identifiés par *RFID* ne sont pas des candidats potentiels, étant connectés à l'Internet des objets par procuration, comme nous en avons déjà parlé dans le Chapitre 2. Cependant, les cas d'utilisations impliquant les objets passifs ne sont pas exclus puisque d'autres objets sont en mesure d'interagir avec eux ; par exemple, une tâche de suivi d'objets dans l'espace pourra être déployée sur les lecteurs *RFID*.

Le réseau ciblé par nos travaux est le réseau Internet, dont l'Internet des objets est supposé être une extension. Au vu de l'évolution technologique et des standards émergents que nous avons présentés au Chapitre 2 (*6LoWPAN*, *CoAP*, etc.), nous supposons que les objets sont capables de communiquer globalement entre eux. Cette hypothèse nous permet de simplifier les aspects du problème de *mapping* de tâche qui portent sur le routage multi-saut, étant donné que le réseau Internet possède déjà une infrastructure de routage d'envergure mondiale.

5.2.1.1 Représentation des coûts

Comme nous l'avons introduit au Chapitre 3, une tâche est représentée par un *graphe logique* $GL = (VL, EL)$ qui exprime quels services continus doivent être répartis dans le réseau et comment ceux-ci communiquent entre eux. Chaque sommet $vl_i \in VL$ est un service continu qui reçoit p flux en entrée et produit q flux en sortie. Le service continu est prévu pour s'exécuter pendant une durée de d^i secondes, typiquement infinie. En outre, on notera IS^i, respectivement OS^i, l'ensemble des flux d'entrée et de sortie d'un service continu vl_i :

$$IS^i = \{S_{ji} \mid (vl_j, vl_i) \in EL\} \text{ et } OS^i = \{S_{ik} \mid (vl_i, vl_k) \in EL\}$$

Pour pouvoir estimer les coûts liés au calcul et à la communication, le débit des flux de sortie doit être déterminé en fonction de chaque famille de services continus :

- Pour les producteurs : déterminé par la fréquence d'échantillonnage (flux d'échantillons) ou par une estimation (flux d'évènements) ;
- Pour les transformateurs : déterminé par le débit des flux d'entrée et la section de travail exécutée ;
- Pour les stockages : déterminé par le débit de production des *streamers*.

En pratique, il peut être difficile de déterminer le débit des flux de sortie d'un transformateur, car il est nécessaire d'exécuter le traitement pour pouvoir mesurer le nombre

d'éléments produits en fonction des entrées. Une solution usuelle consiste à utiliser une approximation, sous la forme d'un *facteur de réduction* [180] :

Définition 5.1 Facteur de réduction Le *facteur de réduction* η^i d'un transformateur vl_i spécifie le nombre d'éléments produits en moyenne par sa section de travail pour chaque élément consommé en entrée. D'où un débit de sortie défini par :

$$\forall S_{ij} \in OS^i, \rho_{ij}(t) = \eta^i \sum_{S_{ki} \in IS^i} \rho_{ki}(t)$$

Concrètement, il est trivial de déterminer analytiquement ce facteur de réduction pour certains services continus : comptage, calcul de la moyenne et projection ($\eta^i = 1$). Toutefois pour des services continus dont la production de résultat est conditionnée par les valeurs des éléments traités (sélection, jointure, etc.), ce facteur de réduction doit être estimé, par exemple en analysant le phénomène mesuré, le domaine des valeurs possibles et la distribution de leurs occurences.

Outre le débit des flux, il est nécessaire de modéliser comment les différents services continus vl_i consomment les ressources matérielles au cours du temps. Dans le contexte du traitement continu, la consommation des ressources évolue en permanence en fonction (i) des variations de débit des flux et (ii) de l'évolution de l'état interne. Ces caractéristiques ne sont pas compatibles avec les approches usuelles de *mapping* de tâche pour les réseaux de capteurs, qui réduisent habituellement la consommation de ressource à des constantes ou à des cycles discrets qui se répètent au cours du temps. Aussi, dans le but d'accroître la précision du processus de *mapping* de tâche, nous décrivons le traitement continu sous la forme de fonctions continues dont la variable est le temps. Il s'agit d'une généralisation des approches classiques, en cela qu'une fonction peut représenter aussi bien l'évolution continue de la consommation des ressources qu'une suite de cycles discrets, si tant est que ces derniers forment une fonction continue par morceaux. En toute logique, nous pouvons compter sur le fait que la consommation des ressources est (i) toujours positive ou nulle et (ii) toujours définie sur l'intervalle de temps considéré.

Définition 5.2 Fonctions de consommation d'espace Les *fonctions de consommation d'espace* $omem^i(t)$ et $odsk^i(t)$ décrivent les besoins en mémoire et en espace persistant d'un service continu au cours du temps, c'est-à-dire entre l'instant t_s correspondant à son déploiement et l'instant courant t. $omem^i(t)$ et $odsk^i(t)$ sont toujours croissantes (non strictement) et exprimées en octets.

Il peut être intéressant d'exprimer ces fonctions à partir du débit $\rho(t)$ ou de la croissance de l'état interne $\sigma^+(x)$, les variations de ce dernier pouvant avoir une influence sur la quantité de mémoire et d'instructions nécessaires pour traiter un nouvel élément en entrée. Par exemple, considérons un transformateur vl_i dont l'état est mis à jour avec chaque nouvel élément d'un flux $S \in IS^i$ et qui ne consomme aucun espace persistant. Soient $\rho(t)$ le débit de S et $\sigma^+(x)$ la fonction de croissance de l'état interne de vl_i,

$omem^i(t)$ et $odsk^i(t)$ peuvent alors être décrites comme suit :

$$omem^i(t) = \sigma^+(\lceil k \rceil) \text{ avec } k = \int_{t_s}^{t} \rho(t)dt$$
$$odsk^i(t) = 0$$

Définition 5.3 Fonction de charge *CPU* La *fonction de charge CPU* $ocpu^i(t)$ décrit le nombre d'instructions requises par seconde (ips) pour exécuter vl_i au cours du temps, conformément aux débits des flux d'entrée.

Définition 5.4 Fonctions de consommation énergétique Les *fonctions de consommation énergétique* $opwr^i_{run}(t)$ et $opwr^i_{com}(t)$ décrivent la quantité d'énergie consommée depuis t_s pour l'exécution du service continu et pour l'émission des flux de sortie. $opwr^i_{run}(t)$ et $opwr^i_{com}(t)$ sont toujours strictement croissantes et sont exprimées en coulomb (C) ou en milliampères-heures (mAh).

Ces deux fonctions dépendent des propriétés de l'objet où le service continu vl_i sera déployé. Si l'on note c^j_{run} le coût d'exécution moyen d'une instruction *CPU* d'un objet n_j et c^j_{com} le coût d'émission moyen d'un octet par n_j, la consommation énergétique se pose comme :

$$opwr^i_{run}(t) = c^j_{run} \int_{t_s}^{t} ocpu^i(t)dt$$
$$opwr^i_{com}(t) = c^j_{com} \sum_{S_{ij} \in OS^i} |S_{ij}(t)|$$

À noter que si cette formulation se base sur des coûts moyens de communication et de calcul, n'importe quel modèle de consommation énergétique peut être utilisé pour produire des estimations plus précises ou pour tenir compte de paramètres spécifiques ; par exemple les mécanismes d'adaptation dynamique de la fréquence et de la tension dans les microprocesseurs (*dynamic voltage and frequency scaling*) [199]. L'Encart 5.1 rappelle trois modèles de consommation énergétique pour la transmission de données dans un réseau sans fil : (i) le *first-order radio model* [196, 200], qui tient compte de la puissance d'émission nécessaire pour compenser la dissipation liée à la distance, (ii) un modèle basé sur les états de l'émetteur (actif, inactif, en transmission) et le coût de transition d'un état à un autre, et (iii) un modèle tenant compte du coût de routage multi-saut. En effet, certaines approches de *mapping* de tâche dans les réseaux de capteurs sans fil considèrent le coût en énergie du routage multi-saut entre les *motes* comme un paramètre majeur [174]. Cela tient du fait que, dans de telles infrastructures, les *motes* peuvent avoir un rôle de routeur en plus de leurs rôles classiques d'acquisition et de calcul ; le routage représentant un coût de communication supplémentaire non négligeable pour chaque *mote*. Toutefois, comme nous l'avons mentionné en début de cette section, nous supposons que c'est l'infrastructure de routage du réseau Internet qui est utilisée. Les routeurs étant alimentés en continu, le coût de communication est donc indépendant

$$E_{elec} \cdot l + \varepsilon_{amp} \cdot l \cdot d^{\chi} \text{ [196]}$$

avec ε_{amp} l'énergie consommée par l'amplificateur (joules par bit par m^2), l le nombre de bits à transmettre, d la distance (mètres) et χ un facteur de dissipation lié à la distance (typiquement $\chi = 2$).

$$\sum_{state} P_{state} \cdot t_{state} + \sum_{trans} P_{trans} \cdot t_{trans} \text{ [197]}$$

avec P_{state} la puissance consommée (watts) par l'émetteur dans un état donné (inactif, actif, actif avec transmission), t_{state} le temps passé dans cet état (secondes), P_{trans} la puissance consommée (watts) par une transition d'état (activation, désactivation, transmission d'un bit) et t_{trans} le temps nécessaire à la transition (secondes).

$$k \cdot \sum_{n_i \in R} E_i \text{ [198]}$$

avec k le nombre de bits à transmettre, $R = (n_1, \cdots, n_N)$ la route suivie par le message à travers N nœuds, et E_i la consommation moyenne en énergie pour transmettre un bit (joules) sur le i-ème nœud de la route (ce modèle tient compte du coût de routage)

Encart 5.1 – Quelques modèles de consommation énergétique pour la communication sans fil.

du nombre de routeurs traversés et seul le coût d'émission des objets vers le premier interlocuteur a une importance.

5.2.1.2 Contraintes de déploiement

Outre la modélisation des ressources consommées, le graphe logique permet d'exprimer des *contraintes* sur les services continus pour influer sur la façon dont ceux-ci seront répartis dans l'environnement d'exécution. Aussi, à un sommet vl_i nous associons un ensemble de contraintes C^i qui indiquent quels sont les prérequis nécessaires pour le déploiement de ce service continu ; par exemple, la présence d'une fonctionnalité particulière ou une propriété de qualité de service que l'on cherche à garantir.

Définition 5.5 Contrainte de déploiement Une *contrainte de déploiement* $c \in C^i$ est décrite sous la forme d'une fonction qui, étant donné la description d'un objet n, indique si la contrainte est validée ou non :

$$c(n) = \begin{cases} 1 & \text{si } n \text{ satisfait la contrainte,} \\ 0 & \text{sinon.} \end{cases}$$

Les contraintes spécifiques prises en charge par *TGCA* portent sur les caractéristiques logicielles et matérielles de l'objet :

Définition 5.6 Contrainte de position Une *contrainte de position* spécifie qu'un service continu doit être déployé sur des objets se trouvant dans une zone donnée. Étant donné une zone géographique A, $l \in \mathbb{L}$ la position de l'objet et $\mathbb{L}$ le domaine de l, on a :

$$c(n) = A(l)$$

Définition 5.7 Zone géographique Une zone géographique est une fonction $A : \mathbb{L} \to \{0, 1\}$ qui, à une position $l \in \mathbb{L}$, indique si l appartient à la zone ou non. Une zone peut être définie par :

- trois points ou plus (polygone) ;
- un centre et un rayon (cercle) ;
- une ou plusieurs étiquettes (p. ex. « bureau-18 ») ;
- une combinaison de zones $a_1 \cdots a_n$, telle que $A(l) = \lceil \sum \frac{a_i}{n} \rceil$.

À noter que le domaine des positions $\mathbb{L}$ varie selon les formats considérés : coordonnées géodésiques, coordonnées sexagésimales, étiquettes, etc.

Définition 5.8 Contrainte de mobilité restreinte Une *contrainte de mobilité restreinte* est une contrainte de position pour objets mobiles, qui spécifie que l'objet doit avoir passé une partie $k \in [0, 1]$ de son existence dans la zone A. Étant donné $(l_1, t_1), \cdots, (l_n, t_n)$ le temps t_i passé à chaque position l_i de l'objet depuis sa mise en service, on a :

$$c(n) = \begin{cases} 1 & \text{si } \sum_i A(l_i)t_i \geq k \sum_j t_j \\ 0 & \text{sinon.} \end{cases}$$

Définition 5.9 Contrainte de mesure Une *contrainte de mesure* spécifie qu'un service continu requiert la présence d'un capteur spécifique possédant une propriété donnée : type sémantique, unité, type concret, fréquence d'échantillonnage, précision, temps de réponse, etc.

Définition 5.10 Contrainte de fonctionnalité Une *contrainte de fonctionnalité* spécifie qu'un service continu requiert une fonctionnalité logicielle ou matérielle donnée : décodage vidéo, chiffrement, etc. Les caractéristiques techniques de ces fonctionnalités peuvent faire partie de la contrainte (format requis, propriété requise). Une fonctionnalité est décrite par un nom et par un ensemble de paires clé-valeur qui représentent les caractéristiques recherchées.

Définition 5.11 Contrainte de qualité de service Une *contrainte de qualité de service* spécifie qu'un service continu doit être déployé sur des objets garantissant un certain niveau de qualité de service (*QoS*). Une propriété de *QoS* possède un nom et une valeur, la contrainte étant relative au domaine de cette valeur : égale, supérieure ou inférieure à un seuil, comprise dans une plage.

Définition 5.12 Contrainte d'adressage Une *contrainte d'adressage* spécifie qu'un service continu peut être déployé uniquement sur un ensemble d'objets caractérisés par leurs adresses.

Un contrainte d'adressage est principalement utilisée pour forcer le déploiement d'un service continu sur un objet précis ou dans une partie connue d'un réseau ; par exemple, un réseau privé.

5.2.1.3 Représentation des objets

Définition 5.13 Réseau Un *réseau*, noté N, est un ensemble d'objets en fonctionnement dans lequel on souhaite déployer des tâches.

Définition 5.14 Objet Un *objet* $n_i \in N$ est un appareil décrit par ses caractéristiques matérielles :

- la quantité de mémoire volatile et persistante, $nmem^i$ et $ndsk^i$, en octets ;
- le nombre d'instructions pouvant être exécutées par seconde par le *CPU*, $ncpu^i$;
- la capacité de la batterie, $npwr^i$, en coulombs ou milliampères-heures ;
- la quantité d'énergie consommée en moyenne pour exécuter une instruction, c^j_{run}, et pour transmettre un octet, c^j_{com} ;
- l'état de ces ressources au démarrage de l'appareil : énergie consommée α_{pwr}, charge *CPU* α_{cpu}, mémoire consommée α_{mem}, espace persistant utilisé α_{dsk} ;
- l'estimation de la consommation énergétique au repos par seconde, β_{pwr} ;
- les métadonnées relatives à l'adresse, la position, les fonctionnalités logicielles et matérielles, les capteurs, et les propriétés de *QoS* de l'objet.

Globalement, de nombreuses métadonnées peuvent être impliquées dans la caractérisation des objets et sont utilisées pour résoudre les contraintes exprimées sur les services continus. En effet, une contrainte n'est *satisfaisable* que si les informations nécessaires sont fournies par la description de l'objet, et ce dans les formats désirés. Aussi, si la description d'un objet n_i ne permet pas de déterminer si une contrainte sur vl_j est satisfaite, n_i sera automatiquement exclu de l'ensemble des candidats potentiels pour l'exécution de vl_j. Qui plus est, en ce qui concerne les fonctionnalités logicielles de l'objet, nous considérons que tout objet est capable d'exécuter un ensemble fixé de traitements simples, tels que le comptage, la projection et la sélection.

Nous introduisons quatre fonctions pour modéliser comment les ressources sont consommées au cours du temps, sachant qu'un objet peut déjà exécuter des services continus au moment où l'on calcule la configuration de déploiement d'une nouvelle tâche. Ces fonctions sont utilisées pour prédire l'évolution de la consommation des ressources et peuvent s'exprimer comme la somme des consommations de chaque service continu

déployé sur n_i. Soit D_o l'ensemble des services continus qui s'exécutent sur n_i, on a :

$$npwr_0^i(t) = \alpha_{pwr} + t \cdot \beta_{pwr} + \sum_{vl_j \in D_o} opwr^j(t)$$

$$ncpu_0^i(t) = \alpha_{cpu} + \sum_{vl_j \in D_o} ocpu^j(t)$$

$$nmem_0^i(t) = \alpha_{mem} + \sum_{vl_j \in D_o} omem^j(t)$$

$$ndsk_0^i(t) = \alpha_{dsk} + \sum_{vl_j \in D_o} odsk^j(t)$$

Ici aussi, il est à noter que des modèles d'agrégation plus élaborés peuvent être utilisés ; par exemple pour tenir compte du coût d'ordonnancement des tâches parallèles et de la commutation de contexte (*context switching*) [201].

5.2.1.4 Recherche d'une configuration

Définition 5.15 Configuration de déploiement Une *configuration de déploiement* pour un graphe logique GL est une matrice M de taille $|N| \times |VL|$ où $m_{ij} = 1$ quand le service continu vl_i est déployé sur l'objet $n_j \in N$, et $m_{ij} = 0$ sinon.

Mesurer la qualité d'une configuration M ne peut pas être effectué lorsque t tend vers l'infini. En effet, les services continus sont supposés être exécutés indéfiniment, et ce même si les flux d'entrée sont limités par des fenêtres. Si l'on part du principe qu'un objet ne peut pas être rechargé, la consommation énergétique est une fonction qui diverge en l'infini, car strictement croissante. En conséquence, évaluer la consommation énergétique lorsque t tend vers l'infini n'a pas de sens du point de vue du problème de *mapping* de tâche. Pour résoudre ce problème, nous introduisons un paramètre pour exprimer la *durée de vie* minimale d'une tâche à déployer.

Définition 5.16 Durée de vie minimale d'une tâche La *durée de vie minimale* δ_l d'une tâche est une contrainte appliquée à tous les services continus de la tâche. Cette contrainte spécifie que la tâche doit pouvoir s'exécuter au moins δ_l unités de temps.

Lorsque les tâches sont concurrentes, nous devons garantir qu'un futur déploiement de tâche n'invalidera pas les garanties de durée de vie spécifiées lors du calcul de M. Cela revient à dire qu'une tâche nouvellement déployée ne doit pas conduire à l'épuisement de l'objet avant que toutes les tâches déjà déployées n'aient été exécutées pour la durée fixée par leurs δ_l respectifs. Pour résoudre ce problème, nous devons donc tenir compte des tâches précédemment déployées, comme c'est déjà le cas dans notre définition des fonctions $n*_0^i(t)$, mais aussi de la durée d'exécution correspondant à la tâche qui se termine en dernier.

Définition 5.17 Durée de vie maximale d'une configuration La *durée de vie maximale* d'une configuration correspond à la durée totale pour laquelle la consommation des

ressources doit être considérée, conformément aux durées de vie minimale des tâches précédemment déployées. Formellement, soit D l'ensemble des tâches déployées, décrites sous la forme de triplets (GL, t_s, δ_l) avec t_s le temps auquel la tâche a démarré et δ_l la durée de vie de la tâche, et $(GL^0, t_s^0, \delta_l^0)$ la nouvelle tâche à déployer. La durée de vie maximale, notée Δ_l, est définie par :

$$\Delta_l = max(\lambda, \delta_l^0)$$

$$\text{avec } \lambda = max(t_s^i + \delta_l^i) - t_s^0, \forall (GL^i, t_s^i, \delta_l^i) \in D$$

Spécifiquement, la qualité d'une configuration se mesure par rapport aux objectifs que l'on désire atteindre, c'est-à-dire (i) minimiser la consommation des ressources de chaque objet et (ii) répartir équitablement les services continus entre les objets. Ces deux objectifs ne sont pas directement compatibles, en cela que la communication entre deux objets peut avoir un coût significativement élevé [67]. On cherche donc à déterminer un optimum entre consommation des ressources et répartition de la charge.

Pour clarifier la formalisation de ces objectifs, nous introduisons quatre fonctions, notées $pwr^i(t), mem^i(t), cpu^i(t)$ et $dsk^i(t)$, pour représenter la consommation globale des ressources d'un appareil n_i. Les fonctions $mem^i(t)$, $dsk^i(t)$ et $cpu^i(t)$ agrègent les consommations de mémoire volatile, d'espace persistant et de *CPU* de tous les services continus vl^j déployés sur n_i. La fonction pwr^i combine l'énergie consommée par tous les services continus pour effectuer leurs calculs, $opwr^j_{run}$, et pour communiquer, $opwr^j_{com}$. Ce dernier terme est ignoré lorsque deux services continus reliés par un flux sont placés sur un même appareil.

$$\begin{aligned}
pwr^i(t) &= \sum_{vl_j \in VL} m_{ji} \cdot \left(opwr^j_{run}(t) + \sum_{S_{jk} \in OS^j} (1 - m_{ki}) \cdot opwr^j_{com}(t) \right) \\
mem^i(t) &= \sum_{vl_j \in VL} m_{ji} \cdot omem^j(t) \\
cpu^i(t) &= \sum_{vl_j \in VL} m_{ji} \cdot ocpu^j(t) \\
dsk^i(t) &= \sum_{vl_j \in VL} m_{ji} \cdot odsk^j(t)
\end{aligned}$$

À partir des objectifs de minimisation de la consommation des ressources et de maximisation de la répartition, nous introduisons quatre fonctions objectifs, notées g_{pwr}, g_{mem}, g_{dsk} et g_{cpu}. Ces fonctions évaluent la consommation de chaque type de ressources – énergie, mémoire volatile, mémoire persistante, processeur – en utilisant les fonctions $pwr^i(t)$, $mem^i(t)$, $dsk^i(t)$ et $cpu^i(t)$. En outre, ces fonctions évaluent la répartition équitable des services continus en mesurant la quantité relative de ressources consommées par un service continu vl_j sur un appareil n_i. Par exemple, si vl_j consomme 20% de la mémoire disponible sur un objet n_1 et 15% de celle d'un objet n_2, alors g_{mem}

donnera un meilleur score à n_2, indépendamment de la quantité de mémoire totale de n_1 et n_2.

$$g_{pwr}(t) = \sum_{n_i \in N} \frac{pwr^i(t)}{npwr^i - npwr_0^i(t_s + t)}$$
$$g_{mem}(t) = \sum_{n_i \in N} \frac{mem^i(t)}{nmem^i - nmem_0^i(t_s + t)}$$
$$g_{dsk}(t) = \sum_{n_i \in N} \frac{dsk^i(t)}{ndsk^i - ndsk_0^i(t_s + t)}$$
$$g_{cpu}(t) = \sum_{n_i \in N} \frac{\int_0^t cpu^i(t)dt}{(t_s + t) \cdot ncpu^i - \int_0^{t_s+t} ocpu_0^i(t)dt}$$

Grâce à ce mécanisme, il est assuré que le pourcentage de ressources consommées est similaire pour chaque objet et que, de fait, les objets les mieux pourvus en ressources ne sont pas systématiquement surchargés. Si, cependant, le comportement de répartition de charge n'est pas souhaité, il suffit d'adapter les fonctions objectifs en conséquence :

$$g_{pwr}(t) = \sum_{n_i \in N} pwr^i(t)$$
$$g_{mem}(t) = \sum_{n_i \in N} mem^i(t)$$
$$g_{dsk}(t) = \sum_{n_i \in N} dsk^i(t)$$
$$g_{cpu}(t) = \sum_{n_i \in N} \int_0^t cpu^i(t)dt$$

5.2.1.5 Résolution exacte

L'optimisation linéaire en nombres entiers [202] est une méthode très populaire pour décrire et résoudre le problème de *mapping* de tâche de manière exacte. Spécifiquement, étant donné que notre configuration M est une matrice de valeurs binaires, nous pouvons représenter la recherche d'une configuration comme un problème d'*optimisation en nombres binaires* (ou *optimisation linéaire 0-1*).

Pour réduire la complexité induite par un problème d'optimisation multiobjectif, nous choisissons d'utiliser une fonction d'agrégation G qui combine les résultats des quatre fonctions objectifs $g_{pwr}(t)$, $g_{mem}(t)$, $g_{dsk}(t)$ et $g_{cpu}(t)$ que nous venons de présenter. Pour la suite, nous utiliserons une fonction d'agrégation consistant en une simple somme, cependant n'importe quelle fonction affine pourrait être utilisée ; par exemple une somme pondérée à partir de constantes a, b, c, d et e :

$$G(t) = a \cdot g_{pwr}(t) + b \cdot g_{mem}(t) + c \cdot g_{dsk}(t) + d \cdot g_{cpu}(t) + e$$

À noter que la formulation actuelle est parfaitement extensible et peut introduire d'autres paramètres si nécessaire, permettant de personnaliser la résolution du problème

avec d'autres caractéristiques (voir Section 5.2.1.6 pour les contraintes relatives à la personnalisation). Aussi, connaissant G, le problème d'optimisation général s'écrit ainsi :

minimiser $G\Big(g_{pwr}(\delta_l), g_{mem}(\delta_l), g_{cpu}(\delta_l), g_{dsk}(\delta_l)\Big)$
tel que $\forall vl_i \in VL, c_k \in C^i, n_j \in N$ et $T = t_s + \Delta_l$

$$\sum_{n_j \in N} m_{ij} = 1 \tag{1}$$

$$npwr_0^j(T) + pwr^j(\delta_l) \leq npwr^j \tag{2}$$

$$nmem_0^j(T) + mem^j(\delta_l) \leq nmem^j \tag{3}$$

$$\int_0^T ncpu_0^j(t)dt + \int_0^{\delta_l} cpu^j(t)dt \leq T \cdot ncpu^j \tag{4}$$

$$ndsk_0^j(T) + dsk^j(\delta_l) \leq ndsk^j \tag{5}$$

$$m_{ij} \leq c_k(n_j) \tag{6}$$

La Contrainte (1) assure que chaque service continu est déployé une seule fois et sur un seul objet. Les Contraintes (2) à (5) garantissent que, pour chaque appareil, les ressources consommées n'excéderont pas les ressources disponibles. En outre, les constantes $npwr^i$, $nmem^i$, $ncpu^i$ et $ndsk^i$ peuvent être remplacées par d'autres constantes positives et inférieures, pour exprimer, par exemple, que l'on souhaite ne jamais consommer plus qu'une portion définie des ressources disponibles (cas typique du *shared sensing*). Enfin, la Contrainte (6) permet de garantir que toutes les contraintes exprimées par chaque service continu sont satisfaisables et satisfaites par l'objet choisi.

Pour l'instant, le problème général que nous venons d'exprimer n'est pas linéaire, c'est-à-dire que toutes les contraintes ne sont pas des fonctions affines. C'est le cas de la contrainte (2) qui fait appel à $pwr^i(t)$ dont la définition donnée précédemment multiplie les variables de décision m_{ji} et m_{ki}. Néanmoins, la fonction $pwr^i(t)$ peut être linéarisée grâce à une technique usuelle consistant à développer le terme non linéaire et à le remplacer par de nouvelles variables de décisions [174]. Dans notre cas spécifique, nous introduisons $|VL|^2 \times |N|^2$ nouvelles variables de décision qui correspondent aux combinaisons possibles pour la multiplication des termes m_{ji} et m_{ki} dans $pwr^i(t)$. Ces nouvelles variables, notées m_{ijkl}, valent $m_{ijkl} = 1$ quand un service continu i est déployé sur un objet j et qu'un service continu k est déployé sur un objet l, et 0 sinon. De là, $pwr^i(t)$ est transformée en :

$$pwr^i(t) = \sum_{vl_j \in VL} \left(m_{ji} \cdot opwr_{run}^j(t) + \sum_{S_{jk} \in OS^j} (m_{ji} - m_{jiki}) opwr_{com}^j(t) \right)$$

Désormais, nous devons ajouter de nouvelles contraintes au problème pour maintenir la cohérence des variables de décision. Les Contraintes (7) et (8) garantissent que m_{ij} ou

m_{kl} ne valent jamais 0 lorsque m_{ijkl} vaut 1. La Contrainte (9), quant à elle, assure que m_{ijkl} ne sera jamais égal à 0 si m_{ij} et m_{kl} valent 1.

$$\forall vl_i, vl_k \in VL, n_j, n_l \in N, m_{ij} - m_{ijkl} \geq 0 \tag{7}$$

$$\forall vl_i, vl_k \in VL, n_j, n_l \in N, m_{kl} - m_{ijkl} \geq 0 \tag{8}$$

$$\forall vl_i, vl_k \in VL, n_j, n_l \in N, 1 + m_{ijkl} \geq m_{ij} + m_{kl} \tag{9}$$

5.2.1.6 Modifications et extensions de la formulation

Notre formulation est conçue de façon à pouvoir être modifiée en fonction des besoins ; par exemple en retirant certains aspects tels que la répartition homogène ou les différents types de coûts (mémoire, calcul, etc.). De la même façon, notre formulation peut être étendue avec des modèles existants capables de représenter des comportements particuliers (modèle de coûts spécifique à une architecture, changement de contexte, multitâche, etc.). Toutefois, deux limites fondamentales sont à considérer :

— le problème doit rester linéaire ;

— les différentes fonctions de coûts doivent être intégrables.

En outre, pour que la fonction d'agrégation G produise des résultats cohérents, l'homogénéité des composantes $g_{pwr}(t)$, $g_{mem}(t)$, $g_{dsk}(t)$ et $g_{cpu}(t)$ doit être assurée. En effet, chaque ressource représentée possède ses propres unités (mAh, coulomb, octets et ips) et ses propres ordres de grandeur (p. ex. la mémoire persistante est généralement disponible en plus grande quantité que la mémoire volatile) d'où la nécessité de garantir une certaine homogénéité avant de les agréger. Notre formulation homogénéise ces informations sous la forme de pourcentages d'utilisation des objets, du fait de la prise en compte de la répartition équitable des services continus à travers le réseau. Cependant, si ces fonctions sont modifiées, par exemple pour ne plus tenir compte de la répartition équitable, alors il est nécessaire de les ramener à une base homogène. Une solution consiste, notamment, à réduire les fonctions de coût à un intervalle $[0, 1]$:

$$g_{pwr}(t) = \sum_{n_i \in N} \frac{pwr^i(t)}{npwr_{max}} \qquad npwr_{max} = \max(npwr_j) \forall n_j \in N$$

$$g_{mem}(t) = \sum_{n_i \in N} \frac{mem^i(t)}{nmem_{max}} \qquad nmem_{max} = \max(nmem_j) \forall n_j \in N$$

$$g_{dsk}(t) = \sum_{n_i \in N} \frac{dsk^i(t)}{ndsk_{max}} \qquad ndsk_{max} = \max(ndsk_j) \forall n_j \in N$$

$$g_{cpu}(t) = \sum_{n_i \in N} \frac{\int_0^t cpu^i(t)dt}{(t - t_s) \cdot ncpu_{max}} \qquad ncpu_{max} = \max(ncpu_j) \forall n_j \in N$$

Si la formulation doit être étendue, les règles de linéarité et d'intégrabilité restent en vigueur. Un exemple intéressant d'extension est celui-ci consistant à assurer le non-dépassement des ressources de calcul non pas lorsque le temps tend vers $T = t_s + \Delta_l$

mais à tout moment. En effet, la contrainte (4) garantit que, une fois le temps T écoulé, l'ensemble des calculs aura pu être effectué ; ce qui ne garantit pas que, étant donné un intervalle de temps quelconque compris entre 0 et T, la quantité d'instructions à exécuter ne dépasse pas la quantité d'instructions pouvant être exécutées. Une telle garantie peut cependant être assurée en transformant la contrainte (4) de la façon suivante :

$$\int_{t_s+ik}^{t_s+(i+1)k} ncpu_0^j(t)dt + \int_{ik}^{(i+1)k} cpu^j(t)dt \leq k \cdot ncpu^j \quad \forall i \in \left[\!\left[0, \frac{\delta_l}{k}\right]\!\right]$$

avec k une constante telle que $k \leq \delta_l$

Concrètement, cette nouvelle contrainte (4) discrétise le temps en intervalles de k unités de temps et garantit qu'à la fin de l'intervalle l'ensemble des calculs aura pu être effectué. À k suffisamment petit, les éventuels dépassements à l'intérieur des intervalles sont négligeables. À noter que la méthode de résolution heuristique que nous présentons ci-après n'est pas soumise aux contraintes de linéarité et d'intégrabilité et pourrait, de fait, remplacer la discrétisation du temps par une recherche du maximum.

5.2.2 Résolution approchée

De manière générale, le problème de *mapping* de tâche est réputé pour être très difficile à résoudre. Notre formulation n'échappe pas à cette règle, car trouver une solution optimale à un problème d'optimisation linéaire en nombres binaires peut s'avérer très coûteux en temps et en calcul [202]. Pour résoudre ce problème, nous proposons une méthode de résolution heuristique avec les objectifs suivants :

- il est possible de trouver une solution approximative en un temps raisonnable ;
- la méthode est suffisamment simple pour être implémentée et exécutée directement par certains objets, typiquement ceux possédant une puissance de calcul moyenne ou supérieure (p. ex. smartphones, micro-ordinateurs à base de *SoC*, etc.).

Ainsi, il n'est pas nécessaire de disposer d'une infrastructure complexe dédiée à la recherche des configurations, et un objet déjà présent dans le réseau peut être élu à cette fin.

5.2.2.1 Algorithme général

Notre algorithme de résolution est un algorithme glouton qui parcourt l'ensemble VL du graphe logique et l'ensemble N des appareils et effectue le choix approprié pour chaque paire de $VL \times N$. Les détails complets sont présentés dans l'Algorithme 5.2, qui se décompose en deux parties distinctes : faisabilité et évaluation.

En premier lieu, pour chaque paire (vl_i, n_j), l'algorithme vérifie si déployer vl_i sur l'objet n_j est *possible*, c'est-à-dire si chaque contrainte de vl_i est satisfaite par n_j ; ressources, position, fonctionnalités logicielles et matérielles requises, qualité de service, etc. Concrètement, il s'agit d'éliminer le plus tôt possible toutes les paires qui ne respectent

Algorithme 5.2 Construction de M

Entrées : N, GL

pour tout $vl_i \in VL$ **faire**
  $val \leftarrow +\infty$
  pour tout $n_j \in N$ **faire**
    si est-possible(vl_i, n_j) **alors**
      $pwrs \leftarrow pwr^i(\delta_l)/(npwr^j - npwr_0^j(T))$
      $mems \leftarrow mem^i(\delta_l)/(nmem^j - nmem_0^j(T))$
      $cpus \leftarrow \int_0^{\delta_l} cpu^i(t)dt/(T \cdot ncpu^i - \int_0^T ocpu_0^i(t)dt)$
      $dsks \leftarrow dsk^i(\delta_l)/(ndsk^j - ndsk_0^j(T))$
      $x \leftarrow G(pwrs, mems, cpus, dsks, ...)$
      si $x < val$ **alors**
        $val \leftarrow x$
        $meilleur\text{-}candidat \leftarrow n_j$
      fin si
    fin si
  fin pour
  si $meilleur\text{-}candidat \neq \varnothing$ **alors**
    $M[vl_i, meilleur\text{-}candidat] \leftarrow 1$
  sinon
    aucune solution
  fin si
fin pour

Algorithme 5.3 est-possible

Entrées : vl_i, n_j

si $npwr_0^j(T) + pwr^i(\delta_l) > npwr^j$
ou $nmem_0^j(T) + mem^i(\delta_l) > nmem^j$
ou $\int_0^T ncpu_0^j(t)dt + \int_0^{\delta_l} cpu^j(t)dt > T \cdot ncpu^j$
ou $ndsk_0^j(T) + dsk^i(\delta_l) > ndsk^j$ **alors**
  retourner faux
fin si
pour tout $c_k \in C^i$ **faire**
  si not $c_k(n_j)$ **alors**
    retourner faux
  fin si
fin pour
retourner vrai

pas les Contraintes (2) à (6) du problème d'optimisation que nous avons formulé plus tôt. Pour ce faire, le prédicat *est-possible* est évalué grâce à l'Algorithme 5.3 qui détermine si la paire est une solution que nous devons soumettre à une analyse plus approfondie.

Aussi, si (vl_i, n_j) est effectivement une solution possible, celle-ci doit être classée par rapport aux autres solutions qui impliquent vl_i, ce pour sélectionner la meilleure option ensuite. La fonction d'évaluation de la qualité d'une solution que nous utilisons ici est similaire à celle définie dans notre formulation, c'est-à-dire basée sur la consommation relative des ressources (p. ex. n_j consommerait $x\%$ de ses ressources pour calculer vl_i). Cependant, la fonction d'agrégation G utilisée par l'algorithme n'est plus obligatoirement restreinte à des fonctions affines ; n'importe quelle fonction pourrait être utilisée (polynôme, fonction par morceaux, escalier, etc.).

La complexité en temps de cet algorithme est $O(|N| \times |VL| \times |C^i|_{max})$, avec $|C^i|_{max}$ le plus grand nombre de contraintes pour un service continu. En effet, pour chaque paire (vl_i, n_j), *est-possible* effectue un nombre d'opérations constantes à l'exception de la vérification des contraintes $c \in C^i$ du service continu. La complexité spatiale est $O(|N| \times |VL|)$, étant donné que seule la matrice M varie en fonction des entrées N et VL. Approximativement, cela nous donne une complexité $O(n^3)$ mais, considérant que le nombre de contraintes $|C^i|$ est plutôt faible en pratique par rapport au nombre de services continus et d'objets, cette complexité est plus proche de $O(n^2)$.

5.2.2.2 Parcours epsilon

Concrètement, l'efficacité de l'algorithme dépend fortement de la manière dont le graphe logique est parcouru. En effet, comme nous l'avons vu, l'algorithme évalue la faisabilité et le coût du déploiement d'une tâche vl_i sur un objet n_j et, une fois trouvé le meilleur choix (local) pour vl_i, ne modifie plus cette association. De fait, l'algorithme de donnera pas les mêmes résultats si le graphe est parcouru aléatoirement ou s'il est parcouru des puits vers les sources. De manière générale, les solutions dont la qualité est très inférieure à celle de la solution optimale émergent lorsque l'algorithme effectue des choix locaux qui s'avèrent mauvais à posteriori. Par exemple, imaginons le déroulement suivant :

1. la tâche vl_i est placée sur l'objet n_j (meilleure solution locale) ;
2. la tâche vl_k, qui est un successeur de vl_i dans le graphe, ne peut pas être placée sur n_j car celui-ci ne possède pas suffisamment de ressources ;
3. la tâche vl_k est placée sur un objet n_l (meilleure solution locale), mais nous devons donc payer le coût de communication entre vl_i et vl_k.

Cette configuration (vl_i, n_j) et (vl_k, n_l) est optimale si, effectivement, ni n_j ni n_l ne peuvent héberger conjointement vl_i et vl_k. Toutefois, si jamais n_l était en mesure d'héberger les deux services continus, alors la configuration obtenue est sous-optimale. Si les coûts d'émission sont très élevés, ce qui est souvent le cas dans les réseaux sans fil [67], alors la solution est fortement sous-optimale.

Aussi, nous cherchons à réduire la possibilité qu'un tel cas de figure se produise avec un parcours de graphe plus sophistiqué. On remarque que, lorsque l'algorithme évalue

Algorithme 5.4 parcours epsilon

Entrées : $GL = (VL, EL)$

calculer ε^{ij} $\forall (vl_i, n_j) \in VL \times N$
tant que tous les vl_i ne sont pas traités **faire**
trouver le vl_i pour lequel ε^{ij} est minimal
vl_i est le prochain sommet à analyser par l'Algorithme 5.2
mettre à jour les ε de tous les successeurs et prédécesseurs de vl_i
fin tant que

la qualité d'une paire (vl_i, n_j), il existe une incertitude relative aux successeurs de vl_i si ces derniers n'ont pas encore été analysés par l'algorithme. En effet, si un successeur vl_j n'est pas défini, alors l'algorithme considère que les deux services continus ne sont pas déployés sur le même objet, c'est-à-dire que $\forall k, m_{jk} = 0$, et intègre donc le coût de la communication dans l'évaluation : il y a surévaluation du coût.

Définition 5.18 Erreur potentielle Soit VL' l'ensemble des services continus non analysés lorsqu'une paire (vl_i, n_j) est évaluée, l'*erreur potentielle* ε^{ij} représente le surcoût engendré dans le pire des cas :

$$\varepsilon^{ij} = \sum_{S_{ik} \in OS^i} |VL' \cap \{vl_k\}| \cdot opwr^i_{com}(\delta_l)$$

Cette surévaluation du coût a un impact sur la façon dont la solution est construite, en cela que l'algorithme va avoir tendance à grouper les services continus sur les premiers objets ayant pris part à la solution (jusqu'à un certain point cependant, du fait des contraintes de répartition équitable). Typiquement, lorsque ε^{ij} est grand, des solutions fortement sous-optimales peuvent être favorisées et nuire à la qualité de la configuration globale. Dans les pires cas, l'algorithme peut ne trouver aucune solution là où il en existe une (faux négatifs), simplement en étant bloqué dans une voie sans issue (c.-à-d. un optimum local à partir duquel une solution globale n'existe plus).

Aussi, nous proposons d'utiliser un algorithme de parcours qui minimise ε^{ij} à chaque itération, c'est-à-dire analyse en premier les vl_i pour lesquels ε^{ij} est minimal. Ce parcours, que nous appellerons *parcours epsilon*, est présenté dans l'Algorithme 5.4. La première étape de l'algorithme consiste à calculer les $|VL| \times |N|$ ε^{ij}. Puis, l'algorithme recherche le sommet vl_i avec le plus petit ε^{ij} ; ce sommet étant ensuite traité par l'Algorithme 5.2. Enfin, les valeurs de ε^{ij} de tous les voisins vl_j de vl_i sont mises à jour et l'algorithme réitère les opérations.

En matière de complexité, l'étape de calcul initial des ε^{ij} et la recherche du vl_i avec le plus petit ε^{ij} s'effectuent toutes deux $|VL| \times |N|$ opérations. Mettre à jour les sommets voisins nécessite, au maximum, de calculer $\Delta(GL) \times |N|$ nouvelles valeurs de ε^{ij} ; $\Delta(GL)$ étant le degré maximal du graphe logique. Sachant que la boucle principale de l'algorithme itère $|VL|$ fois, la complexité totale en temps est de $|VL| \times |N| + |VL| \times (|VL| \times |N| +$

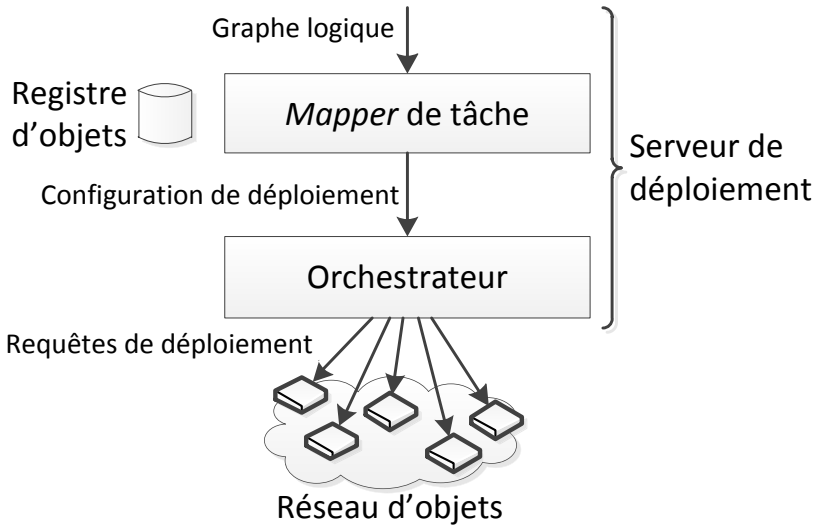


Figure 5.4 – *Mapper* de tâche *TGCA* intégré à *Dioptase*.

$\Delta(GL) \times |N|)$, soit approximativement $O(|VL|^2 \times |N|)$, si l'on considère que $\Delta(GL)$ est toujours très inférieur à $|VL|$. La complexité en espace est $O(|VL| \times |N|)$, car seules les valeurs ε^{ij} sont maintenues en mémoire. En pratique, la complexité peut être réduite en excluant dès le départ les couples $(i, j) \in VL \times N$ qui ne passent pas le test de faisabilité présenté dans l'Algorithme 5.3.

5.3 Implémentation et évaluation

Les algorithmes que nous venons de présenter sont intégrés à *Dioptase* sous la forme d'un module appelé *serveur de déploiement*. Le serveur de déploiement intègre trois composants logiciels majeurs :

- Le *mapper de tâche*, qui implémente les algorithmes que nous avons présentés dans la section précédente. Ce composant est responsable de la transformation des graphes logiques en configuration de déploiement.
- Le *registre d'objets*, qui conserve les descriptions des objets disponibles.
- L'*orchestrateur* qui, étant donné la description d'une configuration, exécute les requêtes de déploiement nécessaires pour déployer les différents services continus sur les objets appropriés.

À partir de ces composants, nous évaluons deux aspects de *TGCA* : (i) l'accroissement de la précision des résultats induits par notre formulation continue et (ii) la qualité des solutions produites par notre algorithme heuristique par rapport à des solutions optimales. Ce second point est étudié pour des graphes logiques générés aléatoirement et pour des graphes logiques relatifs à un problème concret des *smart offices*.

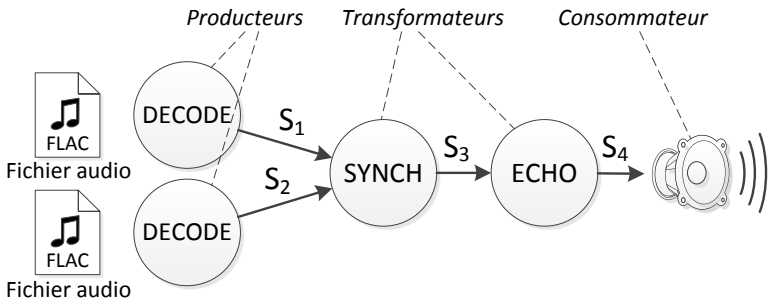


Figure 5.5 – Une tâche de traitement audio.

5.3.1 Évaluation du gain relatif à la modélisation continue

Le premier aspect que nous pouvons évaluer porte sur le bénéfice de la continuité du modèle de consommation des ressources sur la qualité des configurations calculées, en comparaison de l'utilisation de cycles discrets et de constantes (moyennes ou maximum des valeurs).

Pour ce faire, nous nous proposons d'analyser une tâche complexe, présentée sur la Figure 5.5, consistant à (i) décoder deux fichiers audio pour construire deux flux d'échantillons (ou *samples*), (ii) synchroniser les échantillons et (iii) appliquer un simple effet d'écho par duplication et décalage des échantillons. Les données audio sont au format *FLAC*[6], dont le débit est variable en fonction de la complexité du signal original. Cela implique donc que le temps et les ressources nécessaires au décodage d'un cadre (ou *frame*), c'est-à-dire d'un échantillon, fluctuent continuellement ; la consommation est minimale dans le cas d'un échantillon de silence et la consommation est maximale dans le cas d'un échantillon de bruit aléatoire. Ainsi, la consommation des ressources par (i) doit être analysée empiriquement (*profiling*) pour pouvoir en extraire les fonctions qui mesurent ses besoins en calculs, en mémoire et en énergie. À l'inverse, la consommation de (ii) et (iii) est quasi constante, car ces deux services continus sont exécutés sur des échantillons décodés et donc de taille fixe, conformément à la fréquence d'échantillonnage du signal initial.

En pratique, notre analyse porte sur plusieurs exécutions de la tâche sur un smartphone *Galaxy Nexus*[7] (processeur double cœur 1.2GHz ARM Cortex-A9, mémoire 1GB) sur lequel nous avons principalement mesuré la charge du *CPU*, la consommation de mémoire étant comparativement beaucoup plus faible. À partir des mesures, la charge *CPU* continue $ocpu(t)$ est dérivée grâce à une transformée de Fourier discrète (*Discrete Fourier Transform* ou *DFT*). La Figure 5.6 présente l'évolution de la charge *CPU* pour le décodage des deux fichiers audio, obtenue à partir des mesures effectuées sur le smartphone. On peut d'ores et déjà observer le problème général posé par une estimation basée sur la moyenne ou le maximum :

6. *http://xiph.org/flac* (accédé le 08/09/2014).

7. *http://www.samsung.com/us/mobile/cell-phones/SPH-L700ZKASPR* (accédé le 22/09/2014).

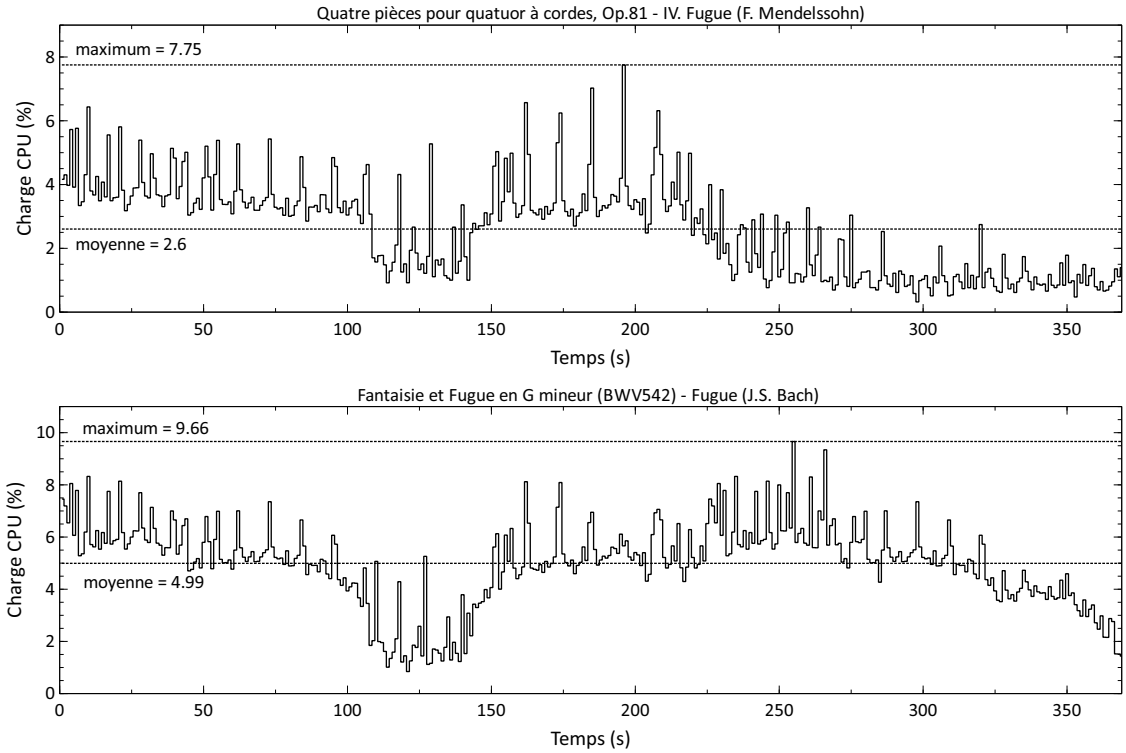


Figure 5.6 – Charge *CPU* pour le décodage de deux fichiers audio.

— si l'on considère le maximum des deux tâches de décodage, la charge *CPU* totale équivaut à $7.75 + 9.66 = 17.41\%$;

— si l'on considère la moyenne des deux tâches de décodage, la charge *CPU* totale équivaut à $2.6 + 4.99 = 7.59\%$.

En réalité, la charge maximale est de 14.76%, à $t = 10s$: potentiellement, l'analyse par les maximums conduit à une surestimation de la charge tandis que l'analyse par les moyennes conduit à une sous-estimation.

En utilisant le modèle que nous avons présenté dans la section précédente pour représenter la consommation des ressources, nous cherchons à estimer :

— combien de fois la tâche complexe (Figure 5.5) peut être exécutée en parallèle sans atteindre 100% de charge *CPU*, étant donné qu'un tel cas de figure introduirait des artefacts pendant la lecture finale ;

— combien de fois cette tâche peut être exécutée consécutivement avant que le smartphone soit à court d'énergie.

Cette estimation est calculée trois fois, avec différentes fonctions $ocpu(t)$: (i) la fonction continue obtenue par *DFT* pour les trois services continus constituant la tâche (décodage, synchronisation et écho), (ii) la fonction constante basée sur la moyenne et (iii) la fonction constante basée sur le maximum. De plus, nous vérifions expérimentalement ces estimations en déployant réellement les services continus sur l'appareil et en mesurant la charge *CPU* ainsi que le temps nécessaire pour épuiser la batterie.

	Moyenne	Maximum	Continu	Expérimental
Tâches parallèles	16	6	8	8.2 $\sigma = 0.48$
Tâches consécutives	17	7	17	16.7 $\sigma = 1.3$

Table 5.2 – Estimation du nombre de tâches déployables sur l'appareil cible.

Les résultats obtenus sont présentés dans la Table 5.2 et montrent que l'estimation basée sur la fonction continue donne des résultats plus précis que les estimations basées sur la moyenne ou le maximum, comme nous pouvions déjà l'observer sur la Figure 5.6. En effet, l'estimation basée sur la moyenne est démesurément optimiste et conduit à des faux positifs (le déploiement aurait échoué avec 16 tâches) tandis que celle basée sur le maximum est trop pessimiste conduisant à des faux négatifs (deux tâches supplémentaires auraient pu être déployées).

5.3.2 Évaluation de l'heuristique pour des problèmes quelconques

Le second aspect que nous pouvons analyser porte sur l'efficacité de l'algorithme heuristique que nous utilisons pour la résolution du problème de *mapping* de tâche : le gain en temps d'exécution et le décalage entre les solutions approchées et les solutions optimales.

Pour illustrer l'efficacité du *parcours epsilon*, nous avons analysé le comportement de l'algorithme avec quatre types de parcours : (i) des sources vers les puits, (ii) des puits vers les sources, (iii) aléatoire (les services continus sont analysés dans n'importe quel ordre) et (iv) parcours epsilon. Pour ce faire, nous générons automatiquement des problèmes quelconques, notés $A \times B$, consistant à rechercher une configuration de déploiement étant donné un graphe logique de A services continus et un réseau de B objets.

Concrètement, le processus de génération sélectionne A services continus, numérotés de 1 à A, depuis un ensemble prédéfini de services continus de complexité variable (constante, linéaire, polynomiale, etc.). Des contraintes sont alors générées aléatoirement pour chaque service continu, à partir d'ensembles prédéfinis de propriétés (zones, fonctionnalités matérielles et logicielles requises, etc.). Enfin, des arêtes sont construites entres les services continus, les cycles étant évités en suivant la règle suivante : $\forall (i, j) \in EL, i < j$.

Les objets B sont eux aussi construits aléatoirement, à partir d'un ensemble de profils prédéfinis (*motes*, systèmes embarqués, smartphones) correspondant à des quantités de ressource bien spécifiques et du même ensemble de propriétés que les contraintes (position, fonctionnalités disponibles, etc.) de façon à s'assurer que le problème puisse être résolu. Les fonctions $npwr_0^i$, $ncpu_0^i$, $nmem_0^i$ et $ndsk_0^i$, qui modélisent la charge courante, sont elles aussi sélectionnées depuis un ensemble de fonctions usuelles (constante, linéaire, polynomiale, etc.).

Les différents problèmes $A \times B$ générés pour cette expérience sont *non triviaux*, c'est-à-dire qu'il est impossible de déployer l'ensemble des A services continus sur un

seul objet. En d'autres termes, la consommation globale du graphe logique, hors coût de communication, est au minimum x fois plus élevée que la quantité d'énergie, de mémoire et d'instructions par seconde disponible sur le plus puissant des appareils.

Les problèmes $A \times B$ générés sont ensuite utilisés en tant que scénario pour le calcul d'une configuration de déploiement. Pour chaque problème, on cherche à calculer :

- une configuration optimale, en résolvant le problème d'optimisation linéaire en nombres binaires que nous avons présenté dans la section précédente ;
- une configuration sous-optimale, en appliquant notre algorithme de résolution heuristique ;
- la plus mauvaise configuration possible (c.-à-d. celle qui consomme le plus d'énergie possible tout en restant déployable), en transformant le problème d'optimisation en une maximisation du coût au lieu d'une minimisation, ce pour mesurer la qualité des solutions sous-optimales vis-à-vis de l'optimal et du pire cas.

Pour le calcul des solutions optimales et des pires solutions, nous utilisons un solveur open source appelé *lp_solve 5.5.2.0*[8], exécuté sur une machine de bureau classique (dual-core 3.2GHz, 4GB memory). Le calcul des solutions sous-optimales est effectué lui aussi sur cette machine, mais aussi sur le smartphone Galaxy Nexus que nous avons utilisé pour notre précédente expérience, dans le but de mesurer l'efficacité de l'algorithme sur ce type d'appareil. Enfin, nous mesurons aussi le nombre de faux négatifs, c'est-à-dire les cas où l'algorithme heuristique ne trouve aucune solution alors que le solveur en trouve une.

La Table 5.3 présente le surcoût moyen de la solution sous-optimale par rapport à l'optimale, ce pour les différents problèmes $A \times B$ que nous avons générés (une centaine de chaque type). Ce surcoût est calculé comme un pourcentage de la valeur de la fonction objectif G pour la solution optimale :

$$\frac{G(sous\text{-}optimal) - G(optimal)}{G(optimal)} \times 100$$

La taille des problèmes est volontairement limitée, du fait du temps de calcul des solutions optimales qui croît démesurément avec la taille ; par exemple, trouver une solution optimale à un problème 50×50 nécessite jusqu'à 5 jours de calculs. Aussi, au-delà des problèmes 15×5 nous ne possédons pas de mesures fiables sur la valeur des solutions optimales. En ce qui concerne le temps d'exécution de l'algorithme heuristique, même pour des problèmes de taille plus élevée, la Table 5.4 présente le temps nécessaire pour la recherche des configurations sous-optimales et, lorsque cela est possible, optimales.

On remarque que, dans la plupart des cas, le parcours des puits vers les sources et le parcours epsilon obtiennent les meilleurs résultats, aussi bien en ce qui concerne les valeurs moyennes que les extremums. Cette efficacité est notamment due au fait que, dans le contexte de la communication sans fil, les coûts d'émissions sont plus élevés

8. *http://lpsolve.sourceforge.net/5.5* (accédé le 22/09/2014).

	Source vers les puits			Puits vers les sources		
	Surcoût	Surcoût min/max	Faux négatifs	Surcoût	Surcoût min/max	Faux négatifs
5×5	18.26	[0, 44.52]	0	8.38	[0, 33.34]	0
5×10	12.66	[0, 39.17]	0	5.51	[0, 21.7]	0
10×5	17.61	[5.55, 56.12]	7	14.05	[0, 42.39]	0
10×10	16.93	[3.38, 37.3]	13	9.82	[0, 25.29]	0
15×5	17.86	[4.94, 51.83]	23	13.47	[0, 39.06]	0
	Epsilon			**Aléatoire**		
	Surcoût	Surcoût min/max	Faux négatifs	Surcoût	Surcoût min/max	Faux négatifs
5×5	8.19	[0, 28.06]	0	25.96	[0, 34.52]	0
5×10	5.42	[0, 21.7]	0	21.87	[0, 24.5]	0
10×5	12.87	[0, 42.39]	0	25.57	[5.01, 45.63]	2
10×10	9.74	[0, 24.77]	0	22.18	[3.61, 23.98]	6
15×5	13.47	[0, 35.27]	0	24.77	[5.27, 36.34]	14

Table 5.3 – Surcoût en pourcentage de l'optimal et nombre de faux négatifs.

	5×5	5×10	10×5	10×10	15×5	20×20	50×50	100×100	1000×1000
Optimal	63ms	1s	8s	60s	483s	>20mn	>20mn	>20mn	>20mn
Heuristique (PC)	<1ms	<1ms	<1ms	1ms	2ms	3ms	6ms	12ms	680ms
Heuristique (smartphone)	<1ms	1ms	2ms	3ms	28ms	101ms	630ms	1s	10s

Table 5.4 – Temps de calcul.

que les coûts de réception. Ainsi, étant donné que le parcours des puits vers les sources déploie les services continus en partant de la fin du graphe logique, l'incertitude ϵ est naturellement réduite lorsque le coût du déploiement d'un vl_i sur un n_j est estimé ; en effet, la plupart des successeurs de vl_i sont déjà associés à un objet.

La Table 5.3 présente, en outre, le nombre de faux négatifs pour chaque méthode de parcours. On constate, ici aussi, que le parcours des puits vers les sources et le parcours epsilon obtiennent de meilleurs résultats en se montrant significativement moins sujets aux faux négatifs. Cela s'explique, de façon similaire, par la réduction implicite (parcours des puits vers les sources) et explicite (parcours epsilon) de l'incertitude. En effet, à chaque étape, ces parcours orientent l'algorithme vers des choix à faible incertitude qui ont donc moins de chance d'être des voies sans issues par la suite.

Spécifiquement, nous pouvons observer que, si les résultats sont proches de l'optimal, il existe toutefois un surcoût qui tend à croître avec la complexité des problèmes, ici

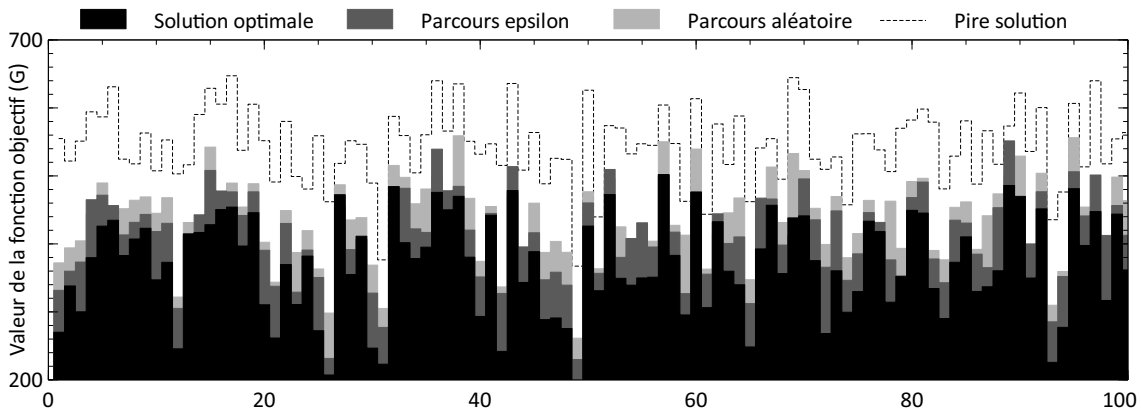


Figure 5.7 – Valeurs de la fonction objectif pour 100 instances d'un problème non trivial (15 opérateurs à placer sur 5 appareils).

traduite par des valeurs de A supérieures à B. Ceci s'observe particulièrement sur la Figure 5.7, qui présente les mesures effectuées lors de la recherche d'une configuration pour les 100 problèmes 15×5 que nous avons générés aléatoirement lors de notre expérience : les valeurs de la fonction objectif pour la solution optimale, la solution sous-optimale obtenue pour le parcours epsilon et le parcours aléatoire et la pire solution possible. Concrètement, ce surcoût est induit par le comportement de l'algorithme heuristique, qui associe graduellement les services continus aux appareils et n'effectue pas de modifications à posteriori. En effet, une fois un vl_i associée à un appareil n_j, l'algorithme ne revient jamais sur vl_i et, de fait, s'oriente très tôt vers une solution dans l'espace de recherche. En conséquence, du fait des forts coûts de communication, la probabilité d'associer un prédécesseur vl_k de vl_i (c.-à-d. $s_{ki} \in IS^i$) sur l'appareil n_j est fortement accrue lorsque vl_i a déjà été associée à n_j. D'où l'influence du parcours sur la qualité des résultats.

Le surcoût des solutions sous-optimales produites par notre algorithme heuristique est cependant à relativiser, car plutôt bas : entre 10 et 20 % en moyenne. De plus, l'algorithme n'est jamais bloqué dans une solution sous-optimale irréalisable (aucun faux négatif) et demeure globalement éloigné de la pire des solutions possibles. Par ailleurs, ce surcoût est à mettre en perspective avec la réduction significative du temps de calcul, même pour des problèmes de taille conséquente (p. ex. 1000×1000) qui seraient impossible à résoudre de manière optimale. Enfin, du fait de la complexité raisonnable de l'algorithme et sa faible consommation d'espace mémoire, celui-ci peut être déployé directement sur un simple smartphone, comme montré dans la Table 5.4. Ainsi, le calcul d'une configuration n'est pas tributaire d'une infrastructure centralisée et peut être réalisé directement sur un appareil courant et massivement répandu dans l'Internet des objets.

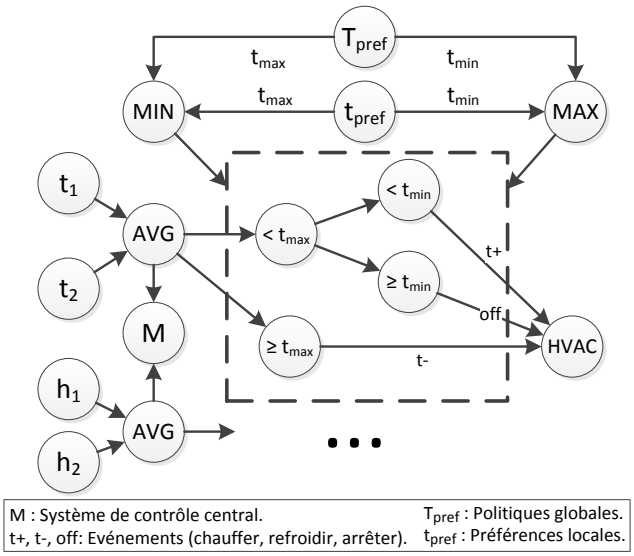


Figure 5.8 – Exemple de graphe logique, pour un bureau contenant deux capteurs de température et deux capteurs d'humidité.

5.3.3 Évaluation de l'heuristique pour un problème concret

Nous venons d'analyser l'efficacité de notre approche heuristique pour un ensemble de problèmes non triviaux dont la structure a été générée aléatoirement. Si ces problèmes aléatoires nous permettent d'obtenir des informations sur le cas général, il est intéressant d'analyser le comportement et l'efficacité de notre algorithme heuristique pour des cas particuliers, c'est-à-dire des cas d'utilisations concrets de l'Internet des objets. Pour ce faire, nous avons choisi le cas d'utilisation dit du « bureau intelligent » (*smart office*) et plus spécifiquement la gestion du système de chauffage, ventilation et climatisation (*CVC*) [9]. Il s'agit d'un problème courant, souvent mentionné comme cas d'utilisation dans la littérature des réseaux de capteurs sans fil et, par extension, dans la littérature de l'Internet des objets [174, 181, 203].

Conformément à ce cas d'utilisation, nous considérons l'existence d'un bâtiment découpée en plusieurs bureaux, chacun d'eux étant équipé d'un climatiseur réversible ainsi que de plusieurs capteurs de température et d'humidité servant à le contrôler. Chaque climatiseur peut être contrôlé localement par l'occupant du bureau, ce dernier disposant d'une interface (p. ex. télécommande ou commande manuelle) pour définir ses préférences en matière de température et d'humidité. En outre, le bâtiment est doté d'un système de contrôle centralisé où un administrateur peut, d'une part, consulter les mesures effectuées par les capteurs de chaque bureau et, d'autre part, définir les politiques globales de gestion du chauffage, de la ventilation et de la climatisation ; par exemple, ne jamais dépasser $27°C$ dans un bureau. Ces politiques sont, par ailleurs, prioritaires sur les préférences locales des utilisateurs.

9. *HVAC* en anglais, pour *Heating, Ventilation and Air-Conditioning*.

Classe de problème	Type d'appareil	Durée de vie (δ_l)	Charge par appareil (%)
simple	*SoC*, smartphones	24 heures	< 10
moyen	systèmes embarqués intermédiaires	90 jours	$[10, 50]$
difficile	systèmes embarqués fortement contraints, *motes*	1 an	> 50

Table 5.5 – Classes de difficulté.

En pratique, chaque bureau contient un certain nombre de capteurs de température et d'humidité, potentiellement embarqués sur un seul et même appareil. Les interactions entre les capteurs et le climatiseur sont assurées par un réseau sans fil, ce même réseau sans fil permettant de communiquer avec le système de contrôle centralisé. Dans ce contexte, on cherche donc à déployer l'ensemble des services continus permettant d'assurer le suivi et le contrôle de la température et de l'humidité des différents bureaux, conformément aux politiques globales et aux préférences locales. Un exemple de graphe logique, correspondant à un bureau équipé de deux capteurs de température, t_1 et t_2, et de deux capteurs d'humidité, h_1 et h_2, est présenté sur la Figure 5.8. Dans cet exemple, les données mesurées depuis les capteurs sont agrégées par un opérateur de calcul de moyenne, *AVG*, puis comparées à un seuil fixé. Ces seuils sont fournis par les opérateurs *MIN* et *MAX* qui font l'arbitrage entre les politiques globales fournies par le système de contrôle central et les préférences locales fournies par le climatiseur. Si les valeurs mesurées sont hors des plages voulues, $[t_{min}, t_{max}]$ et $[h_{min}, h_{max}]$, le climatiseur reçoit l'ordre d'ajuster la température et l'humidité en conséquence. À noter que ce graphe logique comporte volontairement un haut niveau de détail, car une granularité plus fine conduit à une distribution plus fine des services continus dans le réseau et, de fait, une répartition plus équitable.

De manière analogue à l'expérience précédente que nous avons conduite sur les problèmes aléatoires, nous étudions différentes tailles de bâtiment. On notera (A, B) le problème de *mapping* de tâche pour un bâtiment contenant A bureaux équipés de B capteurs d'humidité et de B capteurs de température, dont la fréquence d'échantillonnage est d'une mesure par minute. Conformément à l'exemple de graphe logique présenté sur la Figure 5.8, chaque itération de l'expérience consiste donc à trouver une configuration de déploiement pour $17A + 2B + 3$ services continus vers $2B$ appareils.

En outre, nous cherchons à simuler des problèmes de complexité variable, en jouant notamment sur le temps de vie minimal δ_l de la tâche et le type de systèmes embarqués utilisé pour exécuter les services continus, c'est-à-dire la quantité de ressources disponibles sur chaque appareil. À partir de ces paramètres, nous avons défini trois classes de difficulté, présentées dans la Table 5.5. En effet, nous considérons que le problème CVC est :

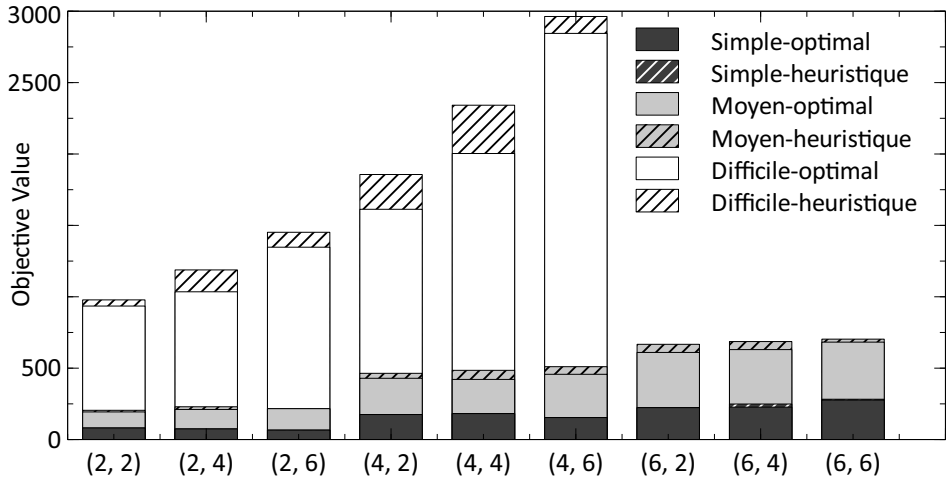


Figure 5.9 – Valeurs de la fonction objectif pour chaque taille et classe de complexité.

Solutions optimales									
	(2, 2)	(2, 4)	(2, 6)	(4, 2)	(4, 4)	(4, 6)	(6, 2)	(6, 4)	(6, 6)
Simple	50ms	120ms	400ms	580ms	4s	18s	19s	83s	727s
Moyen	60ms	150ms	500ms	650ms	5s	23s	26s	92s	932s
Difficile	1822s	2971s	3h	7h	16h	28h	-	-	-
Solutions sous-optimales									
	(2, 2)	(2, 4)	(2, 6)	(4, 2)	(4, 4)	(4, 6)	(6, 2)	(6, 4)	(6, 6)
Simple	<1ms	<1ms	<1ms	<1ms	<1ms	<1ms	<1ms	<1ms	<1ms
Moyen	<1ms	<1ms	<1ms	<1ms	<1ms	1ms	2ms	2ms	2ms
Difficile	<1ms	<1ms	2ms	5ms	5ms	7ms	10ms	12ms	16ms

Table 5.6 – Temps de calcul pour les solutions optimales et sous-optimales.

— simple, pour une durée de vie réduite (24 heures) et un réseau composé de systèmes embarqués puissants tels que des smartphones ou des *Raspberry Pi*[10] ;

— moyennement difficile, pour une durée de 90 jours (3 mois) et un réseau composé d'appareils de capacités moyennes, comme des *Sun SPOT*[11] ;

— difficile, pour une durée de vie élevée (un an) et un réseau composé d'appareils fortement contraints en ressources, ou *motes*, tels que des *Waspmote*[12] ou des systèmes *Arduino*[13].

Nous utilisons ici le même solveur que pour les expériences précédentes, toujours exécuté sur une machine de bureau classique (processeur double cœur 3.2GHz, mémoire

10. *http://www.raspberrypi.org* (accédé le 12/07/2014).

11. *http://sunspotworld.com* (accédé le 12/07/2014).

12. *http://libelium.com* (accédé le 12/07/2014).

13. *http://arduino.cc* (accédé le 12/07/2014).

4GB). Les résultats obtenus pour plusieurs problèmes (A, B) et les trois classes de difficulté sont présentés dans la Table 5.9. Par rapport à nos précédentes expériences sur les graphes logiques générés aléatoirement, nous pouvons constater que le solveur parvient à trouver des solutions optimales pour des graphes de plus grande taille, et ce malgré le fait que le nombre de services continus à déployer soit plus grand que le nombre d'appareils disponibles. En effet, les services continus que nous cherchons à déployer dans le cas réel sont potentiellement moins complexes (moyenne, minimum, maximum, etc.) et, ainsi, l'exploration de l'espace des solutions est plus rapide. Toutefois, lorsque l'on augmente la classe de difficulté, le temps de recherche d'une solution optimale augmente démesurément, comme le montre la Table 5.6.

En ce qui concerne l'écart de qualité entre les solutions optimales et sous-optimales, on observe que celui-ci est inexistant pour les problèmes les plus simples, c'est-à-dire ceux présentant une très faible incertitude potentielle ϵ^i. Pour les autres classes de difficulté, le surcoût de la solution sous-optimale demeure restreint dans les mêmes proportions constatées lors des expériences menées sur les problèmes aléatoires (moins de 20%), ce même pour les problèmes de taille conséquente (par exemple, le problème $(6, 6)$ consiste à déployer 117 services continus sur 12 appareils seulement). Par ailleurs, ici aussi le surcoût modéré est compensé par la réduction importante du temps de calcul.

Chapitre 6

Intégration de réseaux de capteurs tiers ou antérieurs à l'Internet des objets

---

Un grand nombre appareils dotés de capacités de mesure et d'action sont déjà déployés dans l'environnement, notamment les réseaux de capteurs sans fil. Ces systèmes antérieurs ne peuvent pas être, ou difficilement, intégrés directement dans l'Internet des objets au moyen des nouveaux standards que nous avons présentés. Nous proposons donc une solution d'intégration basée sur des *proxys opportunistes mobiles* qui, au cours de leurs déplacements, découvrent les capteurs proches et font le lien avec le reste de l'infrastructure de l'Internet des objets.

Les nouvelles générations d'objets sont capables de communiquer directement sur Internet grâce (i) aux avancées technologiques en matière de systèmes embarqués et (ii) aux protocoles standardisés par l'*IETF* pour les appareils restreints en ressources. Toutefois, l'utilisation de ces protocoles pose de nouveaux problèmes pour les réseaux de capteurs déjà déployés dans l'environnement, étant donné que chaque appareil doit être mis à jour, si possible, ou remplacé. Pour ouvrir ces réseaux de capteurs à l'Internet des objets sans avoir à les modifier, des proxys et des passerelles peuvent être utilisés. Cela implique néanmoins que chaque propriétaire de réseau de capteurs achète, installe et gère ces appareils intermédiaires, ce qui peut être difficile en pratique pour des utilisateurs finaux qui désirent juste contribuer à l'Internet des objets avec leurs capteurs et leurs objets. En outre, certains réseaux de capteurs peuvent avoir été déployés dans des environnements complexes, hostiles ou difficiles d'accès, pour lesquels il n'est pas possible de déployer des proxys à demeure.

Pour résoudre ces problèmes, nous proposons une solution basée sur des *proxys opportunistes mobiles* qui, au cours de leurs déplacements, découvrent les capteurs proches pour (i) les enregistrer auprès des infrastructures de découverte de l'Internet des objets et (ii) exploiter ces capteurs au sein des applications exécutées par un système de traitement de flux, en l'occurrence *Dioptase*. À cette fin, les smartphones modernes sont d'excellents candidats pour agir en tant que proxy, étant donné le grand nombre d'interfaces de communication dont ils sont équipés : *NFC*, *Bluetooth*, *Wi-Fi*/*Wi-Fi Direct* et *3G/4G*. Certains d'entre eux embarquent aussi des interfaces *Zigbee*, comme le *TPH One* [1], ces dernières étant de plus en plus répandues pour les systèmes mobiles [204], notamment dans le cadre des stratégies des constructeurs pour transformer les smartphones et les tablettes en terminaux privilégiés pour l'Internet des objets. De plus, les smartphones sont extrêmement répandus (1,81 milliard dans le monde en 2013) [2] à l'heure actuelle et peuvent couvrir de grandes zones du fait de la mobilité de leurs utilisateurs. Aussi, ce chapitre introduit *Spinel*, un proxy opportuniste pour smartphone.

6.1 Rôles et architecture de *Spinel*

Définition 6.1 Proxy opportuniste Un *proxy opportuniste* est un composant logiciel installé sur un objet mobile, et dont les deux rôles principaux sont :

— découvrir les capteurs statiques qui sont physiquement proches du proxy et faire connaître ces capteurs auprès des infrastructures de l'Internet des objets ;

— présenter les données collectées depuis ces capteurs proches aux applications de l'Internet des objets.

1. *http://www.taztag.com* (accédé le 08/12/2014).

2. *http://www.gartner.com/newsroom/id/2623415* (accédé le 08/12/2014).

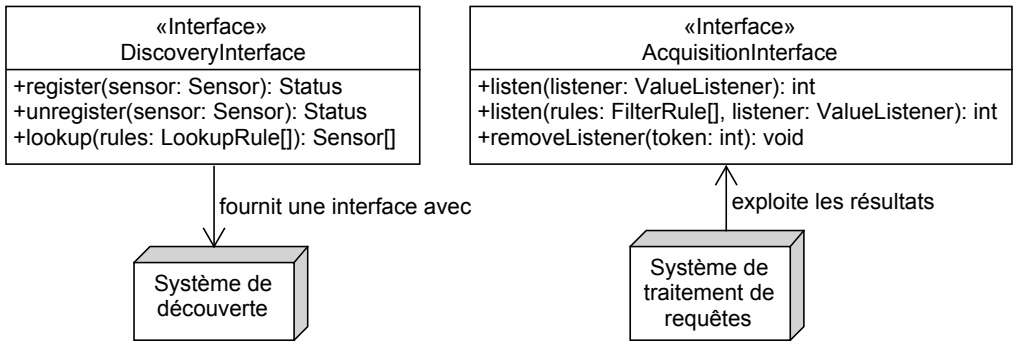


Figure 6.1 – Interfaces de *Spinel* vers les systèmes de l'Internet des objets.

Pour remplir ces deux rôles, un proxy opportuniste présuppose l'existence de deux systèmes existants dans l'infrastructure de l'Internet des objets :

— Un *système de découverte* capable d'enregistrer et de fournir la description d'un ensemble d'objets. Il s'agit d'un composant usuel des architectures orientées service, implémenté sous la forme d'un annuaire global centralisé [46] ou distribué [205].

— Un *système de traitement de requêtes* permettant aux applications de l'Internet des objets d'exprimer des requêtes de traitement de flux distribuées dans un réseau d'objets ; par exemple, *Dioptase*.

Spinel est conçu pour fonctionner en collaboration avec ces systèmes, se concentrant sur les aspects liés à la mobilité et la communication opportuniste. Pour son intégration avec ces systèmes, *Spinel* fournit les interfaces logicielles nécessaires au développeur pour écrire la logique d'interaction avec un système de découverte et un système de traitement de requêtes. Comme présenté sur la Figure 6.1, ces interfaces permettent :

— l'enregistrement et le désenregistrement d'une description de capteur virtuel (adresse, type de données, précision et autres métadonnées) auprès du système de découverte ;

— la recherche de capteurs spécifiques dans le système de découverte, à partir de règles spécifiques à celui-ci ;

— la transmission des mesures acquises par *Spinel* vers le système de traitement des requêtes, au moyen d'un *listener*.

L'architecture globale de *Spinel* est présentée sur la Figure 6.2. *Spinel* exploite le même *pilote d'objet* que *Dioptase* pour gérer les spécificités des différentes classes de capteurs et de plateformes. Ainsi, le développeur peut implémenter l'accès aux capteurs internes et externes sous la forme de *capteurs virtuels*. Comme les smartphones modernes intègrent des *API* spécifiques pour la gestion de capteurs sans fil externes, *Spinel* peut fournir des implémentations génériques de capteurs virtuels ; par exemple pour lire et décoder des mesures produites par des capteurs d'e-santé *Bluetooth* respectant les spécifications *IEEE 11073-104xxx* [206].

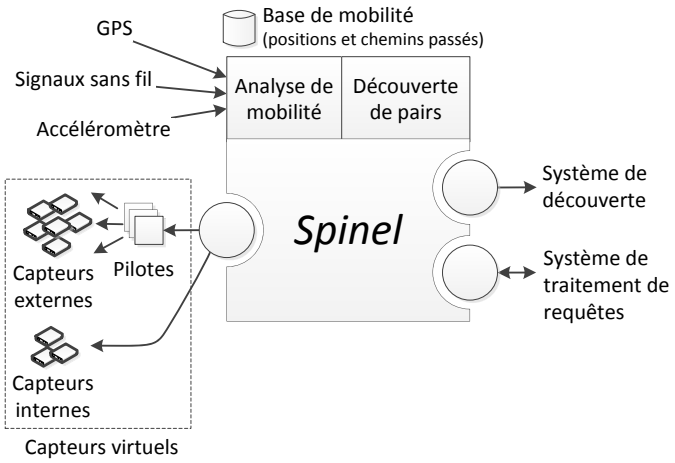


Figure 6.2 – Architecture de *Spinel*.

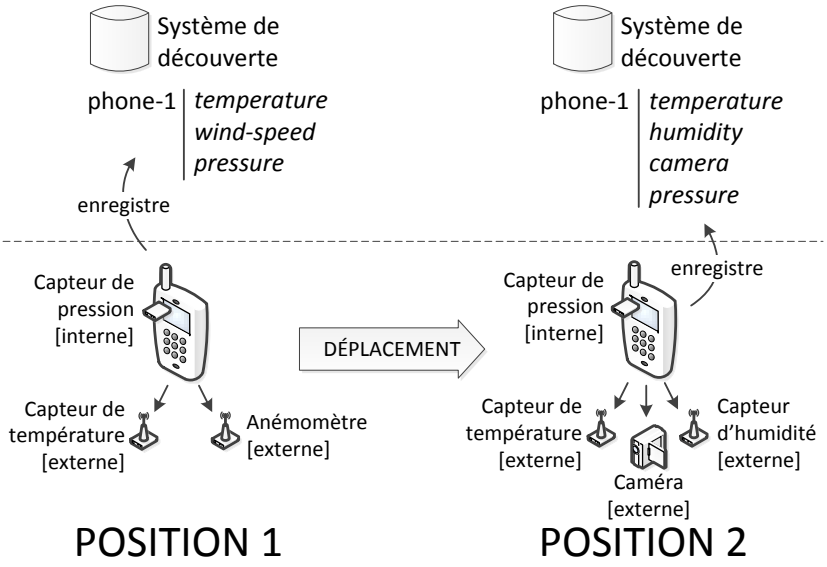


Figure 6.3 – Enregistrement de capteurs internes et externes lors d'un mouvement.

Spinel enregistre les capteurs auprès du système de découverte comme s'ils étaient intégrés au smartphone, qu'il s'agisse de capteurs réellement internes ou de capteurs externes proches, comme illustré sur la Figure 6.3. Ainsi, il n'est pas nécessaire de modifier les systèmes de découverte existants pour fournir la prise en charge des capteurs externes, le système de découverte n'ayant pas connaissance du rôle joué par le proxy. Qui plus est, étant donné que seul le smartphone s'enregistre au lieu d'un ensemble de capteurs distincts, l'espace de recherche du système de découverte est réduit ; le smartphone et l'ensemble des capteurs proches représentant une seule entrée dans le registre. Par exemple, lorsque l'annuaire doit rechercher les capteurs d'une zone donnée, le nombre d'entrées à parcourir est réduit à la liste des smartphones qui se trouvent dans la zone.

La portée théorique maximale des technologies *Bluetooth* et *Zigbee* n'excède pas 100 mètres (50 mètres pour *Bluetooth Low Energy*). Du fait de ces portées relativement faibles et de la mobilité des smartphones, le problème suivant se pose lorsque l'utilisateur marche,

conduit ou utilise les transports en commun : si l'on pose r la portée des capteurs, chaque trajet du proxy de $\frac{r}{2}$ mètres provoque la disparition de 50% des capteurs découverts (hors de portée). Ces capteurs perdus, ainsi que les nouveaux capteurs découverts, doivent alors être notifiés au système de découverte.

Définition 6.2 Effet d'oscillation L'*effet d'oscillation* est un phénomène qui se traduit par une succession de messages d'enregistrements et de désenregistrements envoyés par un proxy opportuniste à un système de découverte. Cet effet a pour conséquence potentielle de surcharger le proxy, le réseau et le système de découverte.

Pour réduire l'effet d'oscillation, *Spinel* exploite un *analyseur de mobilité* dont le but est de déterminer :

— si l'utilisateur est en mouvement, à partir des capteurs de mobilité du smartphone ;

— si l'utilisateur, lorsqu'immobile, est susceptible de se mettre en mouvement, sachant ses positions passées.

Acquérir la position de l'utilisateur au moyen du *GPS* consomme typiquement beaucoup d'énergie [207] et ne peut se faire qu'en extérieur. Sachant que l'utilisateur est volontaire pour héberger le proxy, celui-ci doit être le moins intrusif possible et consommer un minimum de ressources. Pour économiser l'énergie, l'analyseur de mobilité évalue quand et comment exploiter les différents capteurs de mobilité (accéléromètre, *GPS*, etc.) en fonction de leurs coûts et de la précision requise. En outre, l'analyseur de mobilité maintient une base de données des positions et des chemins passés suivis par l'utilisateur au cours du temps : la *base de mobilité*. À partir de l'ensemble de ces informations, l'analyseur de mobilité détermine s'il est pertinent ou non de déclencher le processus de découverte (voir Section 6.2 pour les détails).

Le module de *découverte de pairs* encapsule les différentes *API* de découverte locale fournies par le smartphone pour détecter les appareils proches. Lorsque ce module est exécuté par l'analyseur de mobilité, il démarre les protocoles de découverte pour toutes les interfaces de communication puis notifie *Spinel* de la liste les capteurs disponibles. À partir de cette liste, *Spinel* exploite les interfaces présentées sur la Figure 6.1 pour enregistrer les nouveaux capteurs découverts et notifier le système de traitement de requête des valeurs éventuellement acquises depuis ces capteurs.

Tout comme le prototype de *Dioptase*, le prototype de *Spinel* est implémenté en *Java* et est spécifiquement conçu pour s'exécuter sur la plateforme mobile *Android*. Ce prototype implémente les interfaces nécessaires pour s'intégrer dans une infrastructure composée des systèmes suivants :

— le système de découverte *MobIoT*, un annuaire spécialisé pour l'Internet des objets mobiles, qui permet notamment d'enregistrer des prédictions de mobilité pour les objets (voir Section 6.2 pour des détails) ;

— l'intergiciel de traitement de flux *Dioptase* que nous avons présenté dans les chapitres précédents.

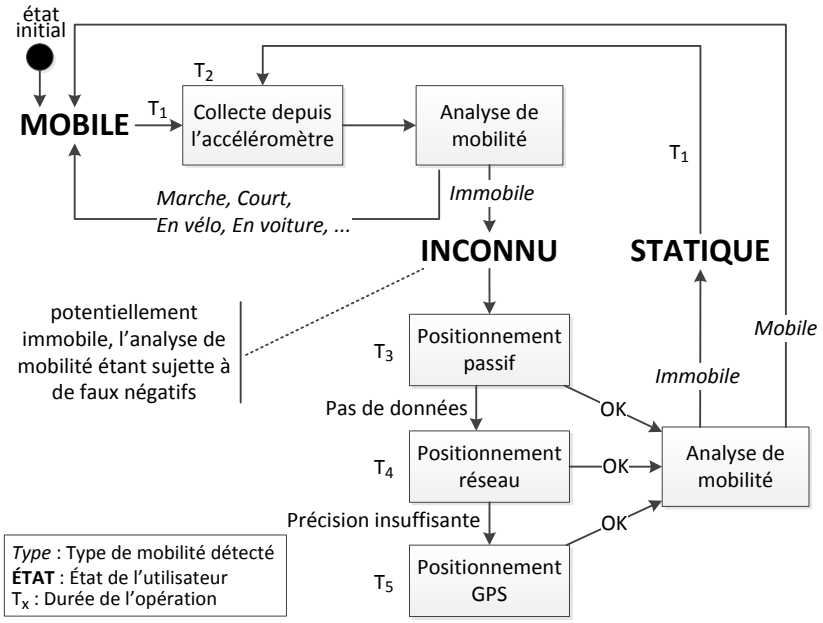


Figure 6.4 – Processus d'analyse de mobilité.

6.2 Analyse et prédiction de mobilité

Spinel analyse constamment les déplacements de l'utilisateur pour pouvoir déterminer quand activer la découverte des capteurs proches et quand enregistrer ces capteurs auprès du système de découverte. En outre, *Spinel* exploite les informations passées pour déterminer à l'avance les trajets de l'utilisateur et préenregistrer les capteurs de façon à réduire le nombre de messages échangés sur le réseau.

6.2.1 Analyse de mobilité

Le processus d'analyse de mobilité fonctionne en trois temps :

— déterminer si l'utilisateur est en mouvement au moyen des capteurs de mobilité les moins coûteux en énergie (détail présenté sur la Figure 6.4) ;

— acquérir la position précise – inférieure à cinq mètres, typiquement fournie par le *GPS* – de l'utilisateur s'il est immobile ;

— déterminer si l'utilisateur va rester à cette position, en considérant ses trajets passés.

Les accéléromètres sont les capteurs de mobilité qui consomment le moins d'énergie sur un smartphone [207]. À partir de leurs mesures, nous inférons l'activité de l'utilisateur (marche, course, immobilité, etc.) et son éventuel mode de transport au moyen de techniques usuelles [208, 209]. La phase d'analyse de l'accéléromètre peut aboutir à deux résultats possibles :

— l'activité réelle de l'utilisateur est déterminée avec exactitude ;

— l'analyse produit un faux négatif, c'est-à-dire détecte que l'utilisateur est immobile tandis qu'en réalité celui-ci est en mouvement.

Contrairement à un faux positif, où l'utilisateur serait considéré en mouvement alors qu'il est immobile, les faux négatifs déclenchent inutilement le processus de découverte et d'enregistrement des capteurs proches. En effet, dans ce cas de figure, la découverte échoue avec une probabilité forte étant donné que les capteurs découverts sont rapidement hors de portée ; la vitesse de marche moyenne d'un être humain étant de 5km/h. Aussi, lorsque l'utilisateur est détecté immobile par l'analyse de l'accéléromètre, nous cherchons systématiquement à confirmer ou infirmer cette information grâce aux autres capteurs de mobilité fournis par le smartphone.

Le détail de l'analyse de mobilité est présenté sur la Figure 6.4. Ce processus est déclenché toutes les T_1 minutes (15 minutes, par défaut) et passe par différentes étapes jusqu'à obtenir une information fiable quant à la mobilité de l'utilisateur. Dans le cas de figure où aucune étape ne fournit une information fiable, l'utilisateur est considéré comme mobile, de façon à minimiser les découvertes inutiles.

La première étape du processus est celle basée sur l'accéléromètre, pour lequel le processus récupère autant d'échantillons que possible pendant T_2 minutes. Cette durée est configurable et doit faire l'objet d'un compromis entre précision et consommation énergétique. Si cette étape ne fournit aucun résultat fiable, le processus passe successivement par les trois modes de détection de la position disponibles sur les systèmes d'exploitation mobiles modernes, par ordre croissant de précision et de coût énergétique :

— le *positionnement passif*, basé sur les positions précédemment acquises par d'autres applications ;

— le *positionnement réseau*, basé sur la force des signaux des cellules *GSM* ou des points d'accès *Wi-Fi* environnants ;

— le *positionnement GPS*, utilisé uniquement en dernier recours si les conditions le permettent.

Tout comme dans le cas de l'accéléromètre, plusieurs échantillons – en l'occurrence des positions – sont collectés pendant des durées prédéterminées, respectivement notées T_3 à T_5. Sachant que le processus est déclenché toutes les T_1 minutes, le temps écoulé entre deux vérifications de mobilité est de $T_1 + T_2$ minutes dans le meilleur des cas, et $T_1 + T_2 + T_3 + T_4 + T_5$ minutes dans le pire cas. Dans tous les cas, si la précision des positions obtenues par les différents capteurs de mobilité est trop faible, l'utilisateur est considéré en mouvement.

Lorsque l'utilisateur est détecté immobile avec certitude, l'analyseur de mobilité récupère ou réutilise la position *GPS* exacte l. Si la base de mobilité contient une ou plusieurs positions proches, la moyenne du temps passé dans la zone est comparée à un seuil pour vérifier s'il est rentable d'effectuer la découverte. Si aucune information n'est disponible dans la base de mobilité, la découverte n'est déclenchée que si la position n'a pas changé depuis les n dernières vérifications.

6.2.2 Prédiction de chemin

Si le processus d'analyse que nous venons d'introduire atténue l'effet d'oscillation, le nombre de messages d'enregistrement auprès du système de découverte peut croître rapidement avec le nombre d'utilisateurs étant donné que :

— tous les proxys d'une même zone découvrent les mêmes capteurs proches (forte redondance) ;

— tous les proxys échangent des messages avec l'annuaire lorsque l'utilisateur est immobile.

La résolution du premier problème est assurée grâce à l'annuaire *MobIoT* [46] qui filtre les appareils possédant des capacités similaires. Cependant, le second problème requiert une méthode pour réduire le nombre de mises à jour transmises au système de découverte. Pour ce faire, nous pouvons exploiter :

— la nature répétable de la mobilité humaine : les individus passent la plupart de leur temps dans un nombre limité de lieux [210] ;

— la capacité de *MobIoT* à enregistrer un objet associé à un trajet plutôt qu'à une position unique.

En pratique, nous pouvons prédire les prochaines positions immobiles de l'utilisateur en nous basant sur sa mobilité passée. Ayant connaissance des capteurs précédemment découverts dans les zones les plus fréquentées par l'utilisateur, *Spinel* peut émettre une seule requête d'enregistrement pour le chemin complet. Ce chemin est alors utilisé par le système de découverte comme un scénario déroulé au cours du temps. Ainsi, le proxy n'a pas besoin d'envoyer de mises à jour au-delà du premier enregistrement, sauf pour corriger la prédiction lorsque celle-ci est erronée.

Définition 6.3 Chemin Un *chemin* $P = (p_1, \cdots, p_n)$ est une suite de n *pauses* $p = (L, a, t, d, S)$ où L est la position (latitude et longitude), a l'azimut, t le temps d'arrivée, d la durée de la pause, et S l'ensemble des capteurs proches découverts à la position L.

Notre technique de prédiction de chemin complète (*Complete Path Prediction*, ou *CPP*) consiste à rechercher un chemin précédemment observé qui correspond aux déplacements les plus récents de l'utilisateur, de façon à prédire sa destination finale et les différentes étapes intermédiaires pour y parvenir. Précisément, étant donné $\mathbb{P}$, un ensemble de chemins décrivant les déplacements passés d'un utilisateur, et P, un chemin composé de ses n dernières positions, *CPP* recherche un chemin $Q \in \mathbb{P}$ qui soit *similaire* à P.

Très peu de travaux dans la littérature abordent ce problème de prédiction de chemin complet par comparaison aux chemins antérieurs, les approches classiques de prédiction étant limitées à la prochaine position de l'utilisateur. À notre connaissance, seul le travail présenté dans [211] s'intéresse à la prédiction de chemin, mais sans répondre aux questions relatives (i) à la sélection des chemins représentatifs de la mobilité de l'utilisateur et (ii) à l'extraction des zones d'intérêts à partir des positions *GPS*.

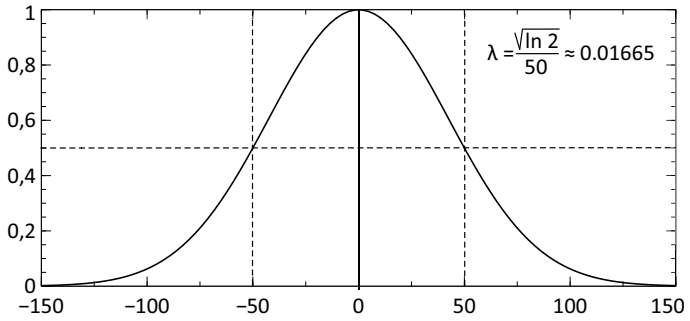


Figure 6.5 – f_d configurée pour indiquer une valeur de similarité de 0.5 lorsque la distance est de 50 mètres : $f_d(50) = 0.5$.

Avant toute chose, nous devons définir la notion de similarité entre deux pauses p et q, en fonction de trois mesures de similarité-dissimilarité :

Définition 6.4 Dissimilarité spatiale La *dissimilarité spatiale* de deux pauses p et q est une valeur correspondant à la distance entre les positions L_p et L_q, notée $\delta(L_p, L_q)$.

Définition 6.5 Dissimilarité temporelle La *dissimilarité temporelle* de deux pauses p et q est un couple de valeurs correspondant à la différence entre les durées et les temps d'arrivées de p et q : $(|t_p - t_q|, |d_p - d_q|)$.

Définition 6.6 Dissimilarité d'azimut La *dissimilarité d'azimut* de deux pauses p et q est une valeur correspondant à la différence d'azimut $|a_p - a_q|$.

Au plus ces valeurs sont élevées, au plus les deux pauses p et q sont dissimilaires. De là, nous définissons la similarité de deux pauses comme suit :

Définition 6.7 Similarité de pauses La *similarité* de deux pauses p et q est une valeur comprise entre 0 (p et q sont *complètement dissimilaires*) et 1 (p et q sont *identiques*), définie par :

$$f_d(\delta(L_p, L_q)) \cdot f_t(|t_p - t_q|) \cdot f_t(|d_p - d_q|) \cdot \left[\cos \frac{|a_p - a_q|}{2}\right]$$

f_d et f_t sont deux fonctions utilisées pour représenter la tendance à découvrir des ensembles de capteurs proches différents lorsque les valeurs de dissimilarité spatiale et temporelle augmentent. f_d et f_t sont des fonctions gaussiennes, de la forme $e^{-(\lambda x)^2}$, qui produisent une valeur comprise entre 0 et 1, comme illustré sur la Figure 6.5.

La similarité entre les pauses étant posée, nous devons considérer la similarité entre deux chemins P et Q. Deux cas doivent être considérés :

- $|P| = |Q|$, auquel cas la similarité est une agrégation des similarités (p_i, q_i) pour tout $0 < i \leq |P|$.
- $|P| \neq |Q|$, auquel cas nous devons sélectionner les couples (p_i, q_j) à évaluer.

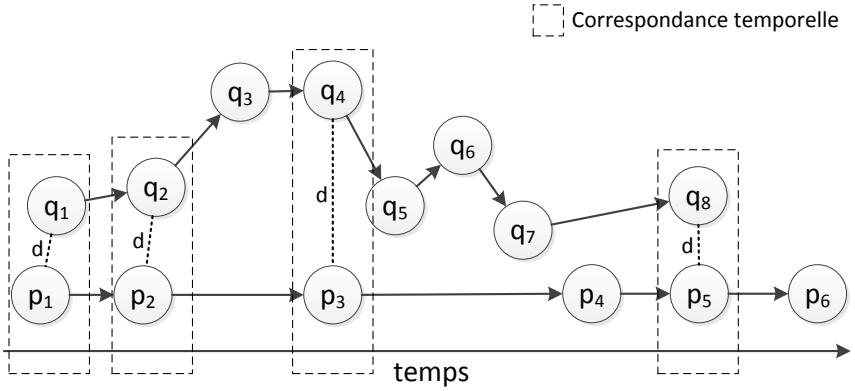


Figure 6.6 – Correspondance temporelle pour la similarité de chemin.

Algorithme 6.1 Calcul des paires (p_i, p_q)

Entrées : P, Q

$X \leftarrow$ matrice nulle de taille $|P| \times |Q|$
pour tout $p_i \in P$ **faire**
  pour tout $q_j \in Q$ **faire**
    $X[i,j] \leftarrow$ **abs**$(t_i - t_j)$
  fin pour
fin pour
pour tout $p_i \in P$ **faire**
  $couple \leftarrow \varnothing, proximite \leftarrow +\infty$
  pour tout $q_j \in Q$ **faire**
    si $X[i,j] < proximite$ **alors**
      $proximite \leftarrow X[i,j], couple \leftarrow (p_i, q_j)$
    fin si
  fin pour
  stocker $couple$
fin pour

Notre stratégie d'appariement couple les pauses qui sont proches dans le temps comme décrit dans l'Algorithme 6.1 et illustré sur la Figure 6.6. Dans cet exemple, la durée totale t_p du chemin P est utilisée comme référence pour la comparaison, ce qui revient à ignorer les sommets de Q dont les temps d'arrivée sont supérieurs à t_p. Ici, la similarité est calculée pour les couples $(p_1, q_1), (p_2, q_2), (p_3, q_4)$ et (p_5, q_8) qui forment l'ensemble des sommets temporellement proche, noté Z. Tous les sommets qui n'appartiennent pas à Z, c'est-à-dire non impliqués dans un couple, sont associés à une valeur constante ; typiquement 0 pour représenter leur dissimilarité avec tous les autres sommets.

Une fois l'appariement effectué, nous sommes en possession de $k = |P| + |Q| - 2|Z|$ valeurs de similarité de pauses $\Omega = \{\omega_1, ..., \omega_k\}$ qui sont agrégées pour former la valeur de similarité des deux chemins P et Q. Nous considérons deux fonctions d'agrégation possibles :

Définition 6.8 Fonction d'agrégation souple La *fonction d'agrégation souple* définit la similarité $\bar{\bar{\mathcal{X}}}(P, Q)$ entre deux chemins P et Q comme la moyenne des valeurs de Ω :

$$\frac{\sum_{i=1}^{k} \omega_i}{k}$$

Définition 6.9 Fonction d'agrégation stricte La *fonction d'agrégation stricte* définit la similarité $\tilde{\tilde{\mathcal{X}}}(P, Q)$ entre deux chemins P et Q comme le produit des valeurs de Ω :

$$\prod_{i=1}^{k} \omega_i$$

La fonction d'agrégation souple a pour effet d'atténuer l'impact des valeurs de similarité faibles par effet de compensation des valeurs faibles par les valeurs fortes. À l'inverse, la fonction d'agrégation stricte implique que deux chemins sont parfaitement dissimilaires dès lors que deux positions sont parfaitement dissimilaires, ce qui est systématiquement le cas si les deux chemins n'ont pas le même nombre de pauses.

Étant donné cette technique de comparaison de chemin, nous introduisons le concept de résolution temporelle :

Définition 6.10 Résolution temporelle La *résolution temporelle* d'un chemin P indique la période de temps couverte par P et la répétabilité de cette période : le jour de la semaine (de lundi à dimanche), le numéro de la semaine dans l'année (de 1 à 53), le jour du mois (de 1 à 31), le mois de l'année (1 à 12), etc.

La résolution temporelle nous permet de rechercher des schémas répétables dans les mouvements passés de l'utilisateur ; par exemple, tous les lundis, toutes les semaines, tous les 3 du mois, tous les février, etc. Au fur et à mesure que l'analyseur de mobilité détermine les pauses de l'utilisateur, un chemin est composé pour les différentes résolutions temporelles actives. À la fin d'un cycle périodique (p. ex. à la fin de la journée ou à la fin du mois), le chemin P ainsi construit est ajouté à la base de mobilité de la façon suivante :

— soit il existe déjà un chemin Q pour lequel $\bar{\bar{\mathcal{X}}}(P, Q)$ est supérieure à un seuil β et les deux chemins sont fusionnés ;

— soit il n'existe aucun chemin Q pour lequel $\tilde{\tilde{\mathcal{X}}}(P, Q)$ est supérieure à un seuil et le chemin est ajouté tel quel.

Lorsqu'un nouveau cycle démarre et à chaque nouvelle pause de l'utilisateur, *Spinel* recherche le chemin passé Q qui correspond le mieux au chemin P suivi par l'utilisateur. S'il existe un maximum $\tilde{\tilde{\mathcal{X}}}(P, Q)$ supérieur à un seuil β, alors *Spinel* l'enregistre auprès du système de découverte. *Spinel* continue ensuite à analyser les mouvements de l'utilisateur et vérifie si la prédiction reste valide à chaque nouvelle pause. Dans le cas où la prédiction serait erronée, c'est-à-dire si la position ou les capteurs proches ne sont pas ceux attendus, *Spinel* envoie une *correction ponctuelle* au système de découverte.

Le pire cas est celui où *Spinel* détecte que le chemin P est une prédiction incorrecte, car un chemin P' correspond mieux aux nouvelles positions. Dans ce cas de figure, *Spinel* effectue une *correction complète* consistant à transmettre l'intégralité de P' au système de découverte. Les corrections complètes ne sont effectuées que si $\widetilde{\widetilde{\mathcal{X}}}(P, P')$ est inférieure à un seuil γ, de façon à éviter une transmission coûteuse lorsque seuls un ou deux sommets sont différents.

6.3 Évaluation de *Spinel*

Au premier démarrage, le prototype de *Spinel* ne possède aucun chemin dans sa base de mobilité et fonctionne automatiquement en mode *collecte pure*. Dans ce mode, le proxy se contente de construire les chemins et contacte le système de découverte à chaque nouvelle pause pour enregistrer les nouveaux capteurs découverts. Lorsque la base de mobilité est suffisamment remplie, c'est-à-dire lorsqu'il existe au moins un chemin pour chaque résolution temporelle considérée, *Spinel* passe en mode *collecte et prédiction*, continuant ainsi à enrichir sa base de mobilité tout en effectuant les enregistrements de chemins auprès du système de découverte. Dans ce mode, au fur et à mesure que l'utilisateur se déplace, le proxy vérifie si la prédiction en cours est toujours valide et la corrige si nécessaire.

Ce comportement étant défini, nous évaluons deux aspects fondamentaux : (i) la consommation énergétique, qui doit être minimisée pour ne pas interférer avec l'usage normal du smartphone, et (ii) la quantité de messages échangés entre le proxy et le système de découverte, ce pour quantifier le bénéfice de notre approche prédictive.

6.3.1 Analyse de la consommation énergétique

Dans notre implémentation, l'analyseur de mobilité et le module de découverte sont deux gros consommateurs potentiels d'énergie. De fait, notre première expérience consiste à analyser leur consommation énergétique pendant une journée, dans un contexte d'utilisation réelle. Pour ce faire, le proxy est déployé sur le smartphone d'un utilisateur et la consommation de la batterie est mesurée pendant quatre jours : (i) sans *Spinel* et (ii) pour trois valeurs différentes de T_1 (le temps écoulé entre deux exécutions du processus d'analyse de mobilité) : 5, 15 et 30 minutes.

Les résultats obtenus sont présentés dans la Figure 6.7 et l'on constate que l'énergie consommée par *Spinel* avec $T_1 = 15$ minutes et $T_1 = 30$ minutes est faible, en comparaison de l'énergie consommée sans *Spinel*. Cependant, la valeur T_1 a un impact sur la structure des chemins construits par *Spinel* étant donné qu'un déclenchement plus fréquent du processus d'analyse de mobilité a tendance à détecter plus de pauses. À titre d'exemple, la Figure 6.8 présente les chemins obtenus pendant l'expérience, chaque chemin représentant le même trajet effectué sur une journée, avec T_1 fixé à 5, 15 et 30 minutes. En considérant que le chemin idéal est celui obtenu à $T_1 = 15$ minutes, nous

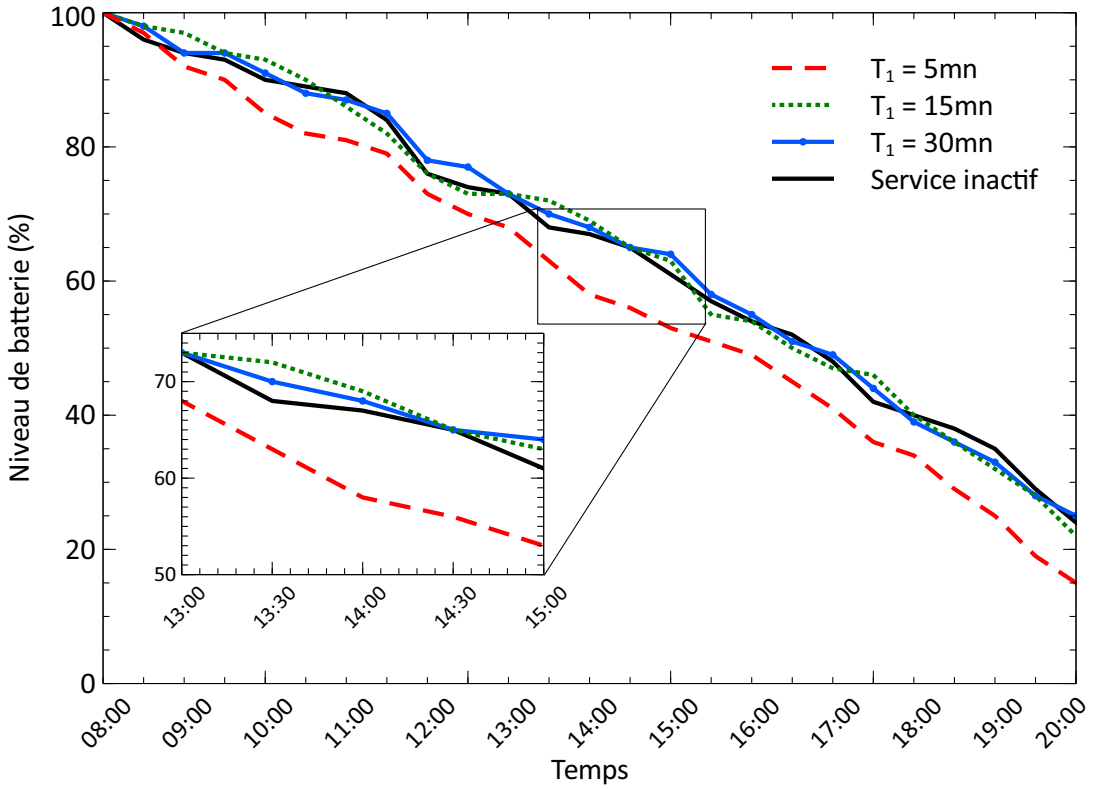


Figure 6.7 – Énergie consommée pendant une journée pour différentes valeurs de T_1.

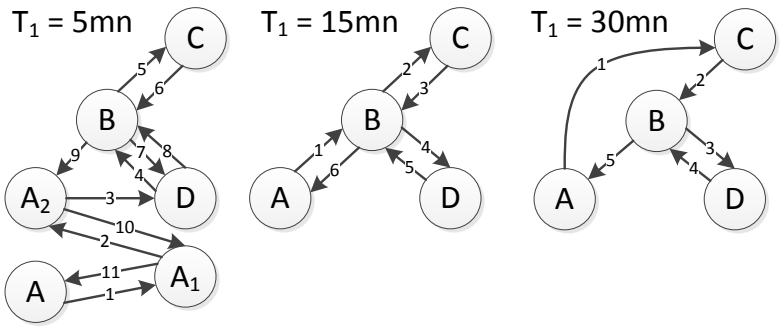


Figure 6.8 – Chemins obtenus pour différentes valeurs de T_1 (durée totale : 1 jour).

observons que, à $T_1 = 5$ minutes, des pauses supplémentaires sont détectées (A_1 et A_2) tandis qu'à $T_1 = 30$ minutes, certaines pauses sont ignorées, notamment l'un des passages par la zone B. Concrètement, le choix de T_1 doit être adapté en fonction du cas d'utilisation et de la précision désirée.

6.3.2 Analyse des échanges réseau

Notre seconde expérience analyse le gain relatif à la prédiction de chemin, c'est-à-dire la réduction du nombre de messages envoyés par le proxy au système de découverte. Dans l'idéal, du fait de la prédiction, le proxy et l'annuaire n'échangent aucune information pendant que l'utilisateur s'éloigne ou s'approche des capteurs connus. Toutefois, si la mobilité humaine tend à être répétable, elle n'est pas pour autant parfaitement prédic-

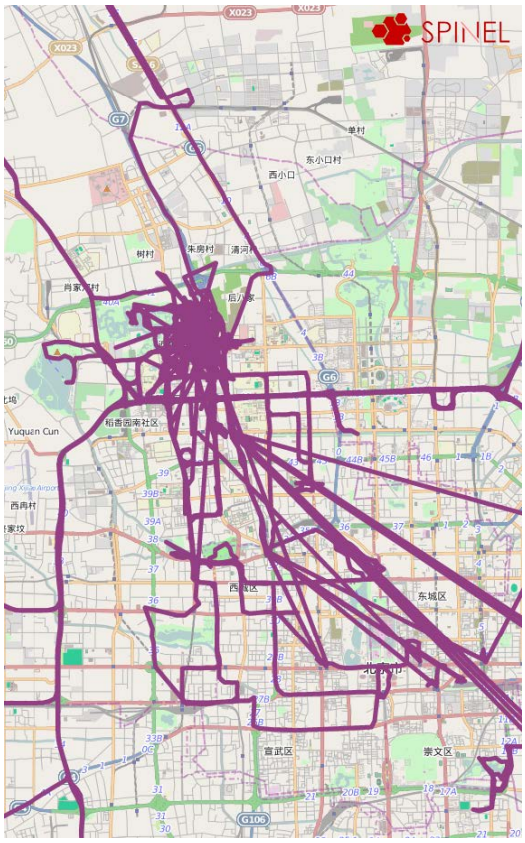


Figure 6.9 – Mobilité de l'utilisateur 001.

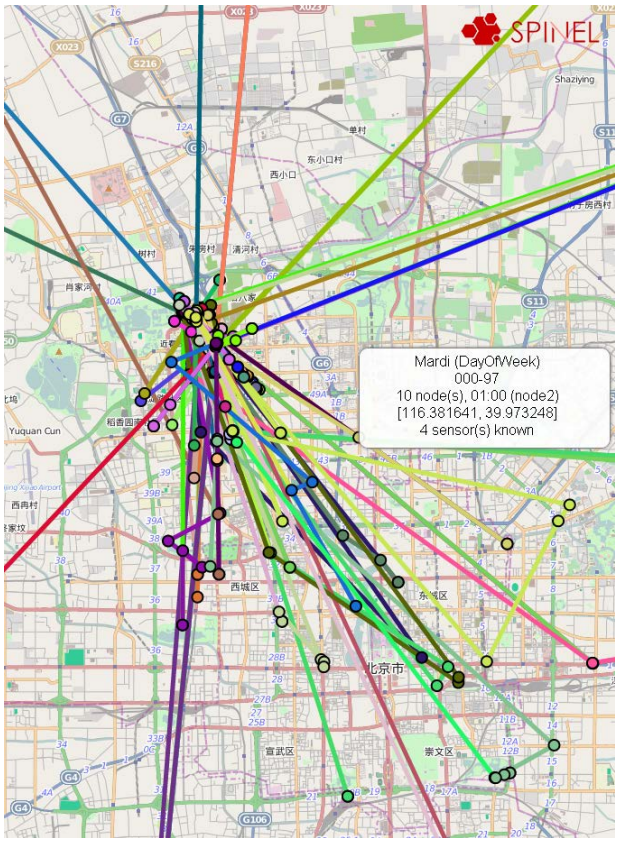


Figure 6.10 – Chemins extraits pour l'utilisateur 001.

tible [212]. De fait, des corrections doivent être envoyées régulièrement au système de découverte. Parfois, la prédiction peut apparaître globalement fausse après plusieurs déplacements ; par exemple lorsque plusieurs chemins de la base de mobilité commencent avec un même sous-chemin. Dans ce cas, la prédiction doit être invalidée auprès du système de découverte et une nouvelle prédiction doit être transmise (correction complète). Ces cas de figure sont coûteux, notamment lorsque la distance entre la position réelle l et la position prédite l' est plus grande que la portée de communication des capteurs. Le seuil β a justement pour rôle d'éviter que *Spinel* enregistre des chemins peu fiables, notamment lorsqu'un nouveau cycle de mobilité démarre.

Dans le but d'évaluer notre approche prédictive dans un environnement comportant beaucoup de capteurs et beaucoup d'utilisateurs, nous optons pour une simulation basée sur : (i) un jeu de données de mobilité déjà existant et (ii) un ensemble de capteurs générés aléatoirement dans la zone considérée. Nous utilisons le jeu de données ouvert dans le cadre du projet *GeoLife*[3] [213] qui implique 182 utilisateurs équipés de *GPS* sur une période de cinq ans. La plupart des 17621 trajectoires enregistrées couvrent la ville de Beijing, Chine. Dans un premier temps, nous retirons les utilisateurs dont la

3. *http://research.microsoft.com/en-us/downloads/b16d359d-d164-469e-9fd4-daa38f2b2e13* (accédé le 10/12/2014).

Algorithme 6.2 Enregistrement prédictif

Entrées : P le chemin correspondant aux n derniers mouvements de l'utilisateur,
$\mathbb{P}$ la base de mobilité,
Q_{-1} la prédiction courante,
S_{-1} l'ensemble des capteurs découvert à la position précédente

$Q \leftarrow \varnothing, meilleur \leftarrow 0$
pour tout $x \in \mathbb{P}$ **faire**
  pour tout $p_i \in P$ **faire**
    $tmp \leftarrow \widetilde{\widetilde{\mathcal{X}}}(x, P)$
    si $tmp \geq \beta$ **et** $tmp > meilleur$ **alors**
      $meilleur \leftarrow tmp, Q \leftarrow x$
    fin si
  fin pour
fin pour
si $Q = \varnothing$ **alors**
  retourner
sinon si $Q \neq Q_{-1}$ **et** $\widetilde{\widetilde{\mathcal{X}}}(Q, Q_{-1}) < \gamma$ **alors**
  effectuer une correction complète
sinon
  découvrir l'ensemble S des capteurs proches
  si $S \neq S_{-1}$ **alors**
    enregistrer $S - (S \cap S_{-1})$
    désenregistrer $S_{-1} - (S \cap S_{-1})$
  fin si
fin si

période d'observation est trop courte (inférieure à un mois) ou fortement discontinue (discontinuités supérieures à une semaine) : le jeu de donnée ainsi nettoyé comporte 64 utilisateurs. Dans un second temps, la mobilité de ces 64 utilisateurs est analysée pour détecter les pauses d'au moins 15 minutes, pour chaque jour de la semaine ; voir la Figure 6.9 pour un exemple de chemins obtenus. Pour chaque utilisateur, la moitié des chemins est utilisée pour entraîner notre système de prédiction et l'autre moitié pour évaluer son efficacité.

En ce qui concerne les capteurs, ceux-ci sont simulés aux environs de divers points d'intérêts (restaurants, supermarchés, parkings, etc.) déterminés grâce au projet de cartographie collaborative *Open Street Map*[4]. En outre, les capteurs sont générés à partir de profils réalistes, comportant des interfaces de communication différentes (*Bluetooth*, *Wi-Fi*, *Zigbee*, etc.) et des portées variables pour simuler les effets d'atténuation que l'environnement peut avoir sur les communications sans fil, notamment dans des zones fortement urbanisées.

4. *http://openstreetmap.org* (accédé le 10/12/2014).

Algorithme 6.3 Enregistrement naïf

Entrées : L la position courante,
L_{-1} la position précédente,
S_{-1} l'ensemble des capteurs découvert à L_{-1}

découvrir l'ensemble S des capteurs proches
si $S \neq S_{-1}$ **alors**
enregistrer $S - (S \cap S_{-1})$
désenregistrer $S_{-1} - (S \cap S_{-1})$
fin si

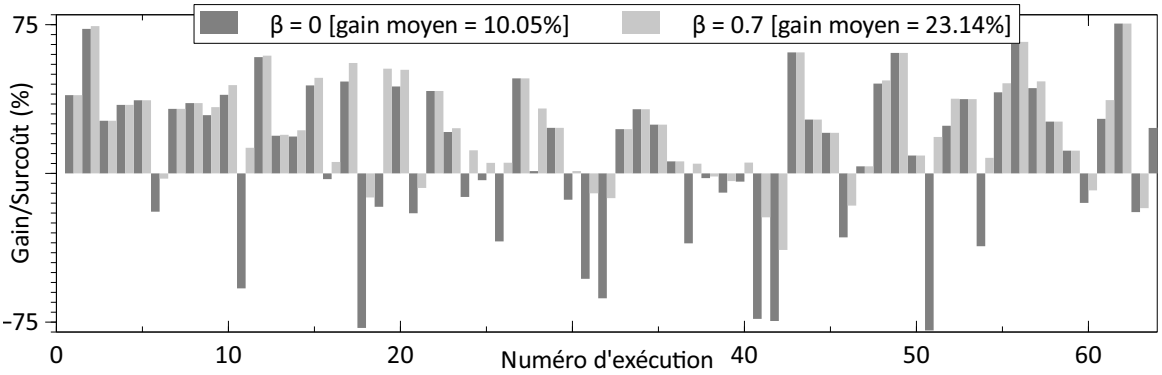


Figure 6.11 – Gain par utilisateur en pourcentage de messages économisés.

Lors de la simulation, nous mesurons le nombre de messages échangés par deux approches d'enregistrement, décrites par les Algorithmes 6.3 et 6.2 :

- l'approche naïve, qui consiste simplement à enregistrer l'ensemble des nouveaux capteurs à chaque nouvelle pause de l'utilisateur ;
- l'approche prédictive, qui se base sur le mécanisme de prédiction de chemin que nous avons décrit précédemment.

Pour chaque nouvelle pause de l'utilisateur, nous mesurons $\mathcal{P}$, le nombre de messages nécessaires pour corriger la prédiction, et $\mathcal{N}$, le nombre de messages envoyés par l'approche naïve. Le gain est alors défini comme :

$$\frac{\mathcal{N} - \mathcal{P}}{\mathcal{N}}$$

Dans certains cas, notamment lorsque la prédiction est totalement erronée ce qui nécessite de retransmettre l'ensemble des capteurs proches, le gain peut être négatif ce qui représente un surcoût par rapport à l'approche naïve.

Cette expérience est répétée pour les 64 utilisateurs présents dans le jeu de données et les résultats obtenus pour chacun d'eux sont présentés sur la Figure 6.11, pour deux valeurs de seuillage : $\beta = 0$ (aucun seuil) et $\beta = 0.7$. Globalement, nous pouvons observer que le gain est supérieur à zéro en moyenne, même lorsque le seuil β n'est pas utilisé (gain d'environ 10%). Comme nous pouvons nous y attendre, l'approche prédictive est sujette à des erreurs de prédiction qui introduisent parfois un surcoût non négligeable ;

jusqu'à 77% pour l'utilisateur 026. Toutefois, ce surcoût est fortement atténué lorsque le seuil β est fixé ; par exemple, à $\beta = 0.7$, nous observons un gain moyen de 23% comparé à l'approche naïve.

6.4 Discussion sur la motivation des utilisateurs

La problématique consistant à motiver les utilisateurs à héberger des proxys opportunistes se retrouve dans de nombreuses applications collaboratives de l'Internet des objets ; par exemple le *sensing* participatif. Dans un premier temps, nos expériences montrent que l'utilisation de *Spinel* ne perturbe pas l'utilisation du smartphone en matière de consommation énergétique, ce qui serait contre-incitatif. Dans un second temps, nous pouvons considérer la littérature existante sur ce problème, notamment au travers (i) de récompenses virtuelles (points, badges, titres, etc.) qui peuvent être comparées avec d'autres utilisateurs ou (ii) de statistiques permettant à l'utilisateur de mesurer son impact positif sur l'environnement et les services rendus aux autres utilisateurs (comportement altruiste).

Concrètement, *Spinel* se prête bien à la *gamification*, notamment car la contribution de l'utilisateur peut être mesurée efficacement :

— durée d'utilisation du proxy et nombre de zones couvertes ;

— nombre de capteurs découverts et nombre de mesures collectées ;

— nombre de requêtes servies avec succès, c'est-à-dire les cas où l'utilisateur est resté à portée des capteurs suffisamment longtemps.

Qui plus est, ces différentes statistiques peuvent être évaluées relativement aux zones fréquentées par l'utilisateur et à ses périodes d'activités récentes, de façon à récompenser spécifiquement les utilisateurs d'une ville ou d'un quartier donné. Cette approche de *gamification* est illustrée par le prototype d'interface présenté sur la Figure 6.12.

Ces différentes statistiques peuvent être exploitées non pas dans un but égocentrique consistant à comparer des scores et des badges entre les utilisateurs, mais dans une dimension altruiste, consistant à montrer quantitativement à l'utilisateur comment son investissement a pu contribuer à améliorer la couverture des capteurs et combien de requêtes ont pu être assurées grâce à lui.

FIGURE 6.12 – Capture d'écran de l'interface de *gamification* de *Spinel*.

Chapitre 7

Conclusion

Notre approche de traitement de flux apporte des solutions à différents problèmes posés par l'Internet des objets, relatifs à la continuité et au volume des données, à l'hétérogénéité des objets et au déploiement automatique de tâches. Cependant, notre travail est une étape vers un Internet des objets répondant à l'ensemble des problématiques que nous avons évoqué dans l'introduction de ce mémoire. Ce chapitre aborde les différentes perspectives de recherche à court et à long terme que nous souhaiterions aborder par la suite.

À l'origine de ce travail de thèse, nous avons posé plusieurs problématiques pour la réalisation de l'Internet des objets en tant que technologie capable de concrétiser à grande échelle divers scénarios relatifs à l'informatique ubiquitaire et à l'intelligence ambiante.

Parmi ces problématiques, nous considérons spécifiquement, d'une part, celles liées à l'architecture, au développement et au déploiement des applications orientées flux pour l'Internet des objets et, d'autre part, celles relatives à la représentation et au traitement des flux de données à grande échelle (spatiale et volumétrique). Pour atténuer le problème de l'échelle, nous proposons :

— de déléguer un maximum d'opérations aux objets eux-mêmes, pour tirer parti de leurs capacités matérielles de plus en plus avancées, sans toutefois négliger les infrastructures de délégation locales (les *surrogates* et les *cloudlets*) et globales (le *cloud*) ;

— d'employer les techniques propres au traitement de flux pour gérer de grands volumes de données dépendantes du temps.

Pour gérer l'hétérogénéité et les spécificités des différents objets dans ce contexte de flux continus, nous revisitons le concept d'architectures orientées service et nous proposons d'introduire une couche intergicielle commune aux différents niveaux hiérarchiques de l'Internet des objets. Concrètement, cet intergiciel permet d'abstraire les objets sous la forme de ressources génériques de mesure, d'action, de calcul et de stockage, reliées entre elles par des flux de données. Sur la base de ces abstractions, nous avons contribué à l'élaboration de nouveaux mécanismes de déploiement de tâche, notamment pour le cas où ces tâches doivent être réparties automatiquement à travers un réseau d'objets (*mapping* de tâche).

Enfin, pour améliorer la protection de la vie privée des utilisateurs et assurer la sécurité des informations et des infrastructures, nous suggérons que l'Internet des objets devrait cloisonner le monde virtuel d'une manière analogue au monde physique ; ces espaces virtuels privés pouvant alors être protégés plus efficacement, notamment en ce qui concerne leurs échanges avec l'extérieur et les services tiers qui composent l'Internet des objets.

7.1 Récapitulatif du travail réalisé et des contributions

Nos différentes contributions gravitent autour de la conception de *Dioptase*, un système de gestion de flux de données pour l'Internet des objets et le Web des objets, permettant de concevoir des applications qui produisent et consomment des flux de données depuis et à destination du monde physique. Dans ce cadre, les contributions spécifiques peuvent être détaillées comme suit.

Une architecture orientée service revisitée spécifiquement pour le traitement de flux : Les données étant produites en continu dans l'Internet des objets, nous avons étudié comment

exploiter une architecture orientée service dans ce contexte de flux potentiellement infinis. À cette fin nous avons proposé une nouvelle architecture mêlant des services discrets manipulant des ensembles finis de données et des services continus manipulant des flux. Dans cette architecture, les services continus produisent des résultats dès lors qu'ils sont instanciés, ces résultats étant accessibles au travers de ports. Nous avons montré par la pratique que cette architecture permettait d'abstraire plusieurs protocoles et techniques de diffusion de flux existants. En outre, nous avons redéfini le principe de composition de service dans ce contexte particulier des flux, grâce à des graphes logiques décrivant le flot des données au travers des services continus.

Un modèle uniforme pour la représentation d'applications orientées flux pour l'Internet des objets : Dans la conception de notre système de gestion de flux de données, notre approche est hybride, combinant la simplicité des *DSMS* génériques avec un cadre théorique fort pour la représentation des flux et de leurs caractéristiques, comme il en existe dans le cadre des *DSMS* issus du modèle relationnel. Ce cadre est suffisamment flexible et extensible pour faire le lien avec les travaux relatifs au Web sémantique pour les réseaux de capteurs et l'Internet des objets, tout en laissant la possibilité aux développeurs d'utiliser leurs propres vocabulaires et technologies de gestion des connaissances. Toutefois, notre approche n'implique pas que les développeurs soient forcés utiliser ce cadre théorique, ces derniers conservant le choix de construire leurs applications en implémentant directement des services continus puis en les reliant entre eux grâce aux connecteurs qui prennent en charge la diffusion des flux. À un niveau supérieur, les développeurs peuvent toutefois exploiter *Dioptase* à son plein potentiel, notamment pour la description des applications sous la forme de graphes logiques combinant des traitements standards issus de la littérature et des traitements décrits avec le langage de traitement de flux *DiSPL*. Grâce aux différents concepts théorisés et unifiés de notre approche de traitement continu, ces applications ainsi décrites peuvent être réparties automatiquement au sein d'un réseau d'objets.

Une approche de mapping *de tâche pour des applications orientées flux dans l'Internet des objets* : Tout comme nous avons revisité une architecture orientée service dans le cadre des flux de données, nous avons formulé un nouveau problème de *mapping* de tâche pour les besoins en traitement continu de l'Internet des objets. Sachant que l'Internet des objets est amené à concrétiser de nombreux scénarios, notre formulation intègre des préoccupations variées qui peuvent être sélectionnées ou non en fonction des scénarios considérés : la réduction de la consommation énergétique globale, la répartition homogène des tâches dans le réseau, la prise en compte des systèmes multitâches ou encore la sélection des objets par rapport à leur position ou leurs capacités matérielles et logicielles. Du fait de la complexité algorithmique de cette famille de problèmes, nous introduisons une méthode de résolution heuristique pour notre formulation spécifique, qui construit itérativement une solution en minimisant les surcoûts potentiels à chaque étape. Nous avons montré

par la pratique que le surcoût assez faible induit par cette approximation est largement contrebalancé par une importante réduction du temps de calcul.

Une méthode d'intégration à l'Internet des objets pour les réseaux de capteurs déjà déployés : L'Internet des objets reposant en partie sur des systèmes existants (réseaux de capteurs et réseaux *RFID*, par exemple), un certain nombre de systèmes sont déjà déployés dans l'environnement, mais sans avoir été préalablement conçus pour interagir avec le réseau Internet. Plutôt que de laisser aux propriétaires de ces réseaux la charge d'acquérir, d'installer et de gérer des proxys pour l'ouverture de leurs appareils à l'Internet des objets, nous proposons une approche basée sur des proxys opportunistes transportés par les utilisateurs de façon à couvrir de larges zones. Ces proxys, installés sur les smartphones, sont conçus pour découvrir les objets proches et faire le lien avec les requêtes extérieures. En outre, des mécanismes d'analyse de mobilité et de prédiction de chemin permettent de déterminer quand agir en tant que proxy, de façon à atténuer l'impact des problèmes relatifs à la mobilité.

En plus de ces contributions scientifiques, notre contribution pratique se résume au prototype de *Dioptase*, capable, à l'heure actuelle, de fonctionner sur un écosystème riche d'appareils. Ce prototype est une base solide pour la réalisation d'un système de traitement de flux de niveau professionnel, exploitable par les développeurs d'applications pour l'Internet des objets. Globalement, le travail conduit dans cette thèse est une étape fondamentale vers d'autres recherches relatives à l'Internet des objets, qui pourront à terme s'appuyer sur *Dioptase* et étendre ses capacités.

7.2 Perspectives futures

Comme nous pouvons l'imaginer, il reste du chemin à parcourir pour parvenir à un Internet des objets répondant à l'ensemble des problématiques que nous avons évoqués dans l'introduction de ce mémoire. À court terme, *Dioptase* est un projet en cours qui peut être amélioré de multiples façons, tandis qu'à long terme nous désirons tendre vers une autonomie accrue des objets et appliquer nos travaux à de grands scénarios pour l'Internet des objets.

7.2.1 Perspectives à court terme

Les perspectives à court terme portent principalement sur l'implémentation du prototype de *Dioptase* qui peut être amélioré de multiples façons.

Intégration des travaux existants : Comme nous l'avons montré dans notre état de l'art, les prédécesseurs de l'Internet des objets sont à l'origine d'une littérature riche, relative à de nombreux sujets tels que la sécurité pour réseaux de capteurs, les opérateurs de traitement continu, l'approximation des flux de données ou encore la gestion de la mobilité et des connectivités intermittentes. Si nous avons intégré une partie de ces travaux dans notre cadre théorique unifié et dans le prototype de *Dioptase*, de nombreux

autres sont à considérer ; opérateurs spécifiques, protocoles de migration, transactions, etc. En outre, nous devons aller plus loin dans l'intégration des réseaux *RFID* et des systèmes de suivi à grande échelle, notamment dans le cadre de l'architecture standardisée à cette fin par le consortium *EPCglobal*.

Amélioration de DiSPL : *DiSPL* est un langage de traitement de flux simple et puissant, qui est pour le moment limité à un jeu d'instructions réduit et une *API* minimale. L'amélioration de *DiSPL*, notamment grâce aux mécanismes d'extensions (bibliothèques), est une perspective intéressante pouvant amener à questionner les primitives de traitement de flux requises pour des usages et des domaines particuliers ; calcul scientifique, calcul vectoriel, apprentissage artificiel, transcodage multimédia, etc. En matière d'implémentation, nos expériences montrent que les traitements interprétés écrits en *DiSPL* sont plus coûteux que leurs homologues écrits en code natif et compilés avec *Dioptase*. Une piste d'amélioration pour les mécanismes d'interprétation de *DiSPL* pourrait consister à introduire un langage intermédiaire compact et plus efficace à interpréter, à l'image du *bytecode* de la plateforme *Java* ou du *Common Intermediary Language* (*CIL*) de la plateforme *.NET*. *DiSPL* serait alors compilé dans ce langage intermédiaire avant d'être déployé sur les objets. De plus, l'utilisation d'un langage intermédiaire peut être un avantage non négligeable pour l'interopérabilité des langages de traitement de flux. En effet, les langages existants peuvent alors être compilés vers cet unique langage intermédiaire, optimisé spécifiquement pour fonctionner directement sur les objets eux-mêmes. Ce type d'approches a déjà été expérimenté dans le cadre des langages généralistes, au travers de la plateforme *LLVM*[1].

Dioptase pour les très petits objets : Des travaux récents ont montré la possibilité de porter des plateformes dynamiques complexes (par exemple, la machine virtuelle *Python*) sur de très petits objets équipés de *Contiki*, un système d'exploitation pour capteurs sans fil. De fait, pour faire de *Dioptase* une solution capable de s'exécuter à tous les niveaux de l'Internet des objets, nous souhaitons étudier dans quelle mesure *Dioptase* et l'interpréteur *DiSPL* peuvent être exécutés sur ces très petits objets. En effet, la modularité de *Dioptase* permettrait de disposer d'une version spécifiquement adaptée aux très petits objets, capable de rendre un minimum de services tout en conservant les interfaces spécifiques à notre architecture orientée service.

7.2.2 Perspectives à long terme

Outre l'amélioration des concepts de *Dioptase* et sa transformation d'un prototype de recherche à une véritable solution professionnelle pour l'Internet des objets, nous voulons poursuivre à long terme notre réflexion sur la conception d'une infrastructure logicielle complète basée sur *Dioptase*. Cela s'inscrit dans notre réflexion globale incluant les différents travaux menés pendant cette thèse : un intergiciel de traitement de flux, un

1. *http://llvm.org* (accédé le 11/12/2014).

système de déploiement centralisé basé sur *TGCA*, un proxy opportuniste pour l'ouverture des réseaux de capteurs existants, une interface d'échange pour la protection des données privées, etc.

À long terme, de nombreux autres composants doivent être considérés, dans la continuité des travaux menés actuellement sur l'Internet des objets. Annuaires, systèmes de nommage, proxys, passerelles, *surrogates*, infrastructures de sécurité, adaptation et raisonnement automatisés, etc. sont autant d'éléments qui sont étudiés à l'heure actuelle, mais qui doivent être intégrés dans la réflexion plus globale de l'Internet des objets en tant que système capable de concrétiser des scénarios. Aussi, nous avons pour perspective à long terme d'exploiter *Dioptase* au sein de scénarios exigeants en matière d'échelle, tels que la ville intelligente ou le *sensing* participatif.

Toujours en lien avec ce qui est défendu dans ce mémoire, la problématique du déploiement dynamique et de l'autonomie des objets demeure fondamentale. Le travail mené sur le problème du *mapping* de tâche, s'il pose un modèle cohérent pour la représentation des caractéristiques des applications et des objets, repose sur une solution statique et centralisée. Notre objectif à long terme consiste à renforcer l'autonomie des objets en étudiant les techniques dynamiques et distribuées pour le *mapping* de tâche, où les différentes tâches migrent entre les objets ; ceux-ci ayant en effet la possibilité de les déléguer à d'autres objets proches. À cette fin, le premier axe à considérer est celui de l'acquisition dynamique de statistiques par les objets, de façon à maîtriser leurs ressources (charge *CPU*, consommation mémoire, consommation énergétique, consommation réseau, etc.) et à analyser leur fiabilité au cours du temps. Ainsi, ces informations pourront être échangées dynamiquement par les objets en pair-à-pair, ou au travers d'autres objets élus dynamiquement, le problème du *mapping* de tâche étant alors réduit à un niveau local représenté par un groupe restreint d'objets. Cela nous amène au second axe, qui correspond aux mécanismes de délégation de tâches interobjets, pour lesquels nous disposons déjà de la couche intergicielle fournie par *Dioptase*. En effet, la possibilité de déployer dynamiquement des tâches interprétées est une étape fondamentale vers un Internet des objets où les appareils autonomes seront en mesure de générer du code *DiSPL* à la volée de façon à diviser et répartir les tâches avec leurs voisins. De la même façon, les objets pourraient s'organiser en fonction des contraintes exprimées sur les tâches qu'ils exécutent et des propriétés qui doivent être maximisées ; par exemple, en dupliquant des tâches ou des parties de tâches au préalable, les objets peuvent plus facilement résister à des incidents relatifs au dysfonctionnement ou à la disparition de certains appareils. Les statistiques collectées localement pourront, en outre, permettre aux objets de déterminer au préalable quels sont leurs homologues les plus fiables pour des opérations données et un niveau de qualité de service requis.

Bibliographie

[1] M. Weiser, "The computer for the 21st century," *Scientific American*, vol. 256, no. 3, 1991.

[2] K. Ashton, "That 'internet of things' thing," *RFID Journal*, 2011.

[3] C. C. Aggarwal, N. Ashish, and A. Sheth, "The internet of things : A survey from the data-centric perspective," in *Managing and Mining Sensor Data*, Springer US, 2013.

[4] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things : Vision, applications and research challenges," *Ad Hoc Netw.*, vol. 10, no. 7, 2012.

[5] INFSO D.4 Networked Enterprise & RFID INFSO G.2 Micro & Nanosystems, in : Co-operation with the Working Group RFID of the ETP EPOSS, "Internet of Things in 2020, Roadmap for the Future," 2008.

[6] "RFID and the inclusive model for the internet of things." *http://www.grifs-project.eu/data/File/CASAGRAS%20FinalReport%20(2).pdf* (accédé le 20/07/2014).

[7] L. Atzori, A. Iera, and G. Morabito, "The internet of things : A survey," *Computer networks*, vol. 54, no. 15, 2010.

[8] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems : The next computing revolution," in *Proceedings of the 47th Design Automation Conference*, DAC '10, 2010.

[9] T. Teixeira, S. Hachem, V. Issarny, and N. Georgantas, "Service oriented middleware for the internet of things : A perspective," in *ServiceWave*, (Poznan, Pologne), Springer-Verlag, 2011.

[10] D. Niyato, E. Hossain, and S. Camorlinga, "Remote patient monitoring service using heterogeneous wireless access networks : architecture and optimization," *Selected Areas in Communications, IEEE Journal on*, vol. 27, no. 4, 2009.

[11] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *Internet of Things Journal, IEEE*, vol. 1, no. 1, 2014.

[12] W. Khan, Y. Xiang, M. Aalsalem, and Q. Arshad, "Mobile phone sensing systems : A survey," *Communications Surveys Tutorials, IEEE*, vol. 15, no. 1, 2013.

[13] A. Kansal, S. Nath, J. Liu, and F. Zhao, "SenseWeb : An infrastructure for shared sensing," *IEEE Multimedia*, vol. 14, no. 4, 2007.

[14] D. Singh, G. Tripathi, and A. Jara, "A survey of internet-of-things : Future vision, architecture, challenges and services," in *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, 2014.

[15] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot) : A vision, architectural elements, and future directions," *Future Gener. Comput. Syst.*, vol. 29, no. 7, 2013.

[16] O. Vermesan and P. Friess, *Internet of things : converging technologies for smart environments and integrated ecosystems.* River Publishers, 2013.

[17] M. A. Feki, F. Kawsar, M. Boussard, and L. Trappeniers, "The internet of things : The next technological revolution," *IEEE Computer Journal*, vol. 46, no. 2, 2013.

[18] S. Deering and R. Hinden, "RFC 2460 - internet protocol, version 6 (IPv6) specification." *http://tools.ietf.org/html/rfc2460* (accédé le 03/09/2014), 1998.

[19] "EPC tag data standard v1.9." *http://www.gs1.org/gsmp/kc/epcglobal/tds* (accédé le 12/12/2014), 2014.

[20] M. Duerst and M. Suignard, "RFC 3987 - internationalized resource identifiers (IRIs)." *http://tools.ietf.org/html/rfc3987* (accédé le 03/09/2014), 2005.

[21] T. Berners-Lee, R. Fielding, and L. Masinter, "RFC 3986 - uniform resource identifier (URI) : Generic syntax." *http://tools.ietf.org/html/rfc3986* (accédé le 03/09/2014), 2005.

[22] P. J. Leach, M. Mealling, and R. Salz, "RFC 4122 - a Universally Unique IDentifier (UUID) URN namespace." *http://tools.ietf.org/html/rfc4122* (accédé le 03/09/2014), 2005.

[23] V. Issarny, N. Georgantas, S. Hachem, A. Zarras, P. Vassiliadist, M. Autili, M.-A. Gerosa, and A. Ben Hamida, "Service-oriented middleware for the future internet : state of the art and research directions," *Journal of Internet Services and Applications*, vol. 2, no. 1, 2011.

[24] P. Mockapetris, "RFC 1034 - domain names – concepts and facilities." *http://tools.ietf.org/html/rfc1034* (accédé le 03/09/2014), 1987.

[25] H.-D. Ma, "Internet of things : Objectives and scientific challenges," *Journal of Computer Science and Technology*, vol. 26, no. 6, 2011.

[26] I. Ishaq, D. Carels, G. K. Teklemariam, J. Hoebeke, F. V. d. Abeele, E. D. Poorter, I. Moerman, and P. Demeester, "IETF standardization in the field of the internet of things (IoT) : a survey," *Journal of Sensor and Actuator Networks*, vol. 2, no. 2, 2013.

[27] L. Mottola and G. P. Picco, "Programming wireless sensor networks : Fundamental concepts and state of the art," *ACM Computing Survey*, vol. 43, no. 3, 2011.

[28] M. Nottingham and R. Sayre, "RFC 4287 - the atom syndication format." *http://tools.ietf.org/html/rfc4287* (accédé le 03/09/2014), 2005.

[29] C. C. Aggarwal, "Mining sensor data streams," in *Managing and Mining Sensor Data*, Springer US, 2013.

[30] H. Andrade, B. Gedik, and D. Turaga, *Fundamentals of Stream Processing : Application Design, Systems, and Analytics.* Cambridge University Press, 2014.

[31] A. B. Sharma, L. Golubchik, and R. Govindan, "Sensor faults : Detection methods and prevalence in real-world datasets," *ACM Trans. Sen. Netw.*, vol. 6, no. 3, 2010.

[32] K. Ni, N. Ramanathan, M. N. H. Chehade, L. Balzano, S. Nair, S. Zahedi, E. Kohler, G. Pottie, M. Hansen, and M. Srivastava, "Sensor network data fault types," *ACM Trans. Sen. Netw.*, vol. 5, no. 3, 2009.

[33] C. C. Aggarwal and J. Han, "A survey of RFID data processing," in *Managing and Mining Sensor Data*, Springer US, 2013.

[34] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom, "A pipelined framework for online cleaning of sensor data streams," tech. rep., EECS Department, University of California, Berkeley, 2005.

[35] A. L. Carreton, K. Pinte, and W. De Meuter, "MORENA : A middleware for programming nfc-enabled android applications as distributed object-oriented programs," in *Proceedings of the 13th International Middleware Conference*, Middleware '12, (New York, NY, USA), Springer-Verlag New York, Inc., 2012.

[36] M. Zorzi, A. Gluhak, S. Lange, and A. Bassi, "From today's INTRAnet of things to a future INTERnet of things : a wireless- and mobility-related view," *Wireless Communications, IEEE*, vol. 17, no. 6, 2010.

[37] D. Arsíc, B. Schuller, and G. Rigoll, "Suspicious behavior detection in public transport by fusion of low-level video descriptors," in *Multimedia and Expo, 2007 IEEE International Conference on*, 2007.

[38] R. H. Weber, "Internet of things – new security and privacy challenges," *Computer Law & Security Review*, vol. 26, no. 1, 2010.

[39] L. Golab and M. T. Özsu, "Data stream management," *Synthesis Lectures on Data Management*, vol. 2, no. 1, 2010.

[40] N. A. Chaudhry, K. Shaw, and M. Abdelguerfi, *Stream data management*, vol. 30. Springer, 2005.

[41] C. Bockermann, "A survey of the stream processing landscape," tech. rep., TU Dortmund University, 2014.

[42] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB : an acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, 2005.

[43] G. Amato, S. Chessa, and C. Vairo, "MaD-WiSe : a distributed stream management system for wireless sensor networks," *Software : Practice and Experience*, vol. 40, no. 5, 2010.

[44] A. Brayner, A. L. Coelho, K. Marinho, R. Holanda, and W. Castro, "On query processing in wireless sensor networks using classes of quality of queries," *Information Fusion*, vol. 15, 2014.

[45] M. Papazoglou, "Service-oriented computing : concepts, characteristics and directions," in *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, 2003.

[46] S. Hachem, A. Pathak, and V. Issarny, "Service-oriented middleware for large-scale mobile participatory sensing," *Pervasive and Mobile Computing*, vol. 10, no. 1, 2014.

[47] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio, "Interacting with the SOA-based internet of things : Discovery, query, selection, and on-demand provisioning of web services," *Services Computing, IEEE Transactions on*, vol. 3, no. 3, 2010.

[48] S. Hachem, A. Pathak, and V. Issarny, "Probabilistic registration for large-scale mobile participatory sensing," in *Proc. of the 11th IEEE International Conference on Pervasive Computing and Communications*, PerCom '13, 2013.

[49] B. Billet, G. Bouloukakis, N. Georgantas, S. Hachem, V. Issarny, M. Autili, D. Di Ruscio, P. Inverardi, T. Massimo, A. Di Salle, D. Athanasopoulos, P. Vassiliadis, and A. Zarras, "D1.4 final CHOReOS architectural style and its relation with the CHOReOS development process and IDRE," tech. rep., CHOReOS Consortium, 2013.

[50] B. Billet and V. Issarny, "From task graphs to concrete actions : A new task mapping algorithm for the future internet of things," in *Proc. of the the 11th IEEE International Conference on Mobile Ad hoc and Sensor Systems*, MASS '14, 2014.

[51] B. Billet and V. Issarny, "Dioptase : A distributed data streaming middleware for the future web of things," *Journal of Internet Services and Applications*, vol. 5, no. 13, 2014.

[52] B. Billet and V. Issarny, "Spinel : An opportunistic proxy for connecting sensors to the internet of things (under submission)," 2015.

[53] H. Zimmermann, "OSI reference model – the ISO model of architecture for open systems interconnection," *IEEE Transactions on Communications*, vol. 28, no. 4, 1980.

[54] N. Gershenfeld, R. Krikorian, and D. Cohen, "The internet of things," *Scientific American*, vol. 291, no. 4, 2004.

[55] D. Guinard and V. Trifa, "Towards the web of things : Web mashups for embedded devices," in *Proceedings of the International World Wide Web Conferences*, 2009.

[56] Z. Song, A. Cárdenas, and R. Masuoka, "Semantic middleware for the internet of things," in *Internet of Things (IOT), 2010*, 2010.

[57] E. Welbourne, L. Battle, G. Cole, K. Gould, K. Rector, S. Raymer, M. Balazinska, and G. Borriello, "Building the internet of things using RFID : The RFID ecosystem experience," *Internet Computing, IEEE*, vol. 13, May 2009.

[58] N. Prabhu, *Design and Construction of an RFID-enabled Infrastructure : The Next Avatar of the Internet.* CRC Press, 2013.

[59] R. Want, "Enabling ubiquitous sensing with RFID," *Computer*, vol. 37, no. 4, 2004.

[60] R. Want, "An introduction to RFID technology," *Pervasive Computing, IEEE*, vol. 5, no. 1, 2006.

[61] J. Banks, D. Hanny, M. Panchano, and L. G. Thompson, *RFID Applied.* Wiley, 2007.

[62] "EPC radio-frequency identity protocols generation-2 UHF RFID v2.0.0." *http://www.gs1.org/gsmp/kc/epcglobal/uhfc1g2* (accédé le 09/07/2014), 2013.

[63] H. Alemdar and C. Ersoy, "Wireless sensor networks for healthcare : A survey," *Computer Networks*, vol. 54, no. 15, 2010.

[64] T. Watteyne and K. Pister, "Smarter cities through standards-based wireless sensor networks," *IBM Journal of Research and Development*, vol. 55, no. 1.2, 2011.

[65] D. Basu, G. Moretti, G. Gupta, and S. Marsland, "Wireless sensor network based smart home : Sensor selection, deployment and monitoring," in *Sensors Applications Symposium (SAS), 2013 IEEE*, 2013.

[66] V. Dyo, S. A. Ellwood, D. W. Macdonald, A. Markham, N. Trigoni, R. Wohlers, C. Mascolo, B. Pásztor, S. Scellato, and K. Yousef, "WILDSENSING : Design and deployment of a sustainable sensor network for wildlife monitoring," *ACM Trans. Sen. Netw.*, vol. 8, no. 4, 2012.

[67] P. A. C. d. S. Neves and J. J. P. C. Rodrigues, "Internet protocol over wireless sensor networks, from myth to reality," *Journal of Communications*, vol. 5, no. 3, 2010.

[68] "IEEE standard for local and metropolitan area networks – part 15.4 : Low-rate wireless personal area networks (LR-WPANs)," 2011.

[69] "SIGFOX flyer." *http://www.sigfox.com/fr/download/fichiers/118/Flyer-SIGFOX-v1.2.pdf* (accédé le 12/08/2014), 2003.

[70] A. Dunkels, J. Eriksson, L. Mottola, T. Voigt, F. J. Oppermann, K. Römer, F. Casati, F. Daniel, G. P. Picco, S. Soi, S. Tranquillini, P. Valleri, S. Karnouskos, P. Spieß, and P. M. Montero, "D-1.1 – Application and Programming Survey," tech. rep., MakeSense Consortium, 2010.

[71] I. F. Akyildiz and I. H. Kasimoglu, "Wireless sensor and actor networks : research challenges," *Ad Hoc Networks*, vol. 2, no. 4, 2004.

[72] M. Di Francesco, S. K. Das, and G. Anastasi, "Data collection in wireless sensor networks with mobile elements : A survey," *ACM Trans. Sen. Netw.*, vol. 8, no. 1, 2011.

[73] A. Dunkels, J. Alonso, T. Voigt, H. Ritter, and J. Schiller, "Connecting wireless sensornets with TCP/IP networks," in *Wired/Wireless Internet Communications*, Springer Berlin Heidelberg, 2004.

[74] N. Kushalnagar, G. Montenegro, and C. Schumacher, "RFC 4919 - IPv6 over low-power wireless personal area networks (6LoWPANs) : Overview, assumptions, problem statement, and goals." *http://tools.ietf.org/html/rfc4919* (accédé le 03/09/2014), 2007.

[75] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "RFC 4944 - transmission of IPv6 packets over IEEE 802.15.4 networks." *http://tools.ietf.org/html/rfc4944* (accédé le 03/09/2014), 2007.

[76] T. Winter, P. Thubert, T. Clausen, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, and J. Vasseur, "RFC 6550 - RPL : IPv6 routing protocol for low power and lossy networks." *http://tools.ietf.org/html/rfc6550* (accédé le 03/09/2014), 2012.

[77] Z. Shelby, K. Hartke, and C. Bormann, "RFC 7252 - the constrained application protocol (CoAP)." *http://tools.ietf.org/html/rfc7252* (accédé le 03/09/2014), 2014.

[78] M. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. Grieco, G. Boggia, and M. Dohler, "Standardized protocol stack for the internet of (important) things," *Communications Surveys Tutorials, IEEE*, vol. 15, no. 3, 2013.

[79] D. Wang, D. Evans, and R. Krasinski, "IEEE 802.15.4J : extend IEEE 802.15.4 radio into the MBAN spectrum [industry perspectives]," *Wireless Communications, IEEE*, vol. 19, no. 5, 2012.

[80] "ZigBee-2007 specification." *http://www.zigbee.org/Specifications.aspx* (accédé le 12/08/2014), 2007.

[81] J. Hui and P. Thubert, "RFC 6282 - compression format for IPv6 datagrams over IEEE 802.15.4-based networks." *http://tools.ietf.org/html/rfc6282* (accédé le 03/09/2014), 2011.

[82] R. Fielding and J. Reschke, "RFC 7230 - hypertext transfer protocol (HTTP/1.1) : Message syntax and routing." *http://tools.ietf.org/html/rfc7230* (accédé le 03/09/2014), 2014.

[83] E. Wilde *et al.*, "Putting things to REST," 2007.

[84] K. Kenda, C. Fortuna, A. Moraru, D. Mladenić, B. Fortuna, and M. Grobelnik, "Mashups for the web of things," in *Semantic Mashups*, Springer Berlin Heidelberg, 2013.

[85] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle, "The web of things : Interconnecting devices with high usability and performance," in *Embedded Software and Systems, 2009. ICESS '09. International Conference on*, 2009.

[86] L. Richardson and S. Ruby, *RESTful web services.* " O'Reilly Media, Inc.", 2008.

[87] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim, "TinyREST : A protocol for integrating sensor networks into the internet," in *Proc. of REALWSN*, 2005.

[88] F. Delicato, P. Pires, L. Pinnez, L. Fernando, and L. da Costa, "A flexible web service based architecture for wireless sensor networks," in *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, 2003.

[89] I. Corredor Pérez and A. M. Bernardos Barbolla, "Exploring major architectural aspects of the web of things," in *Internet of Things*, Springer International Publishing, 2014.

[90] S. Nastic, S. Sehic, M. Vogler, H.-L. Truong, and S. Dustdar, "PatRICIA – a novel programming model for iot applications on cloud platforms," in *Service-Oriented Computing and Applications (SOCA), 2013 IEEE 6th International Conference on*, 2013.

[91] T. Lin, H. Zhao, J. Wang, G. Han, and J. Wang, "An embedded web server for equipment," in *Parallel Architectures, Algorithms and Networks, 2004. Proceedings. 7th International Symposium on*, 2004.

[92] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle, "Smews : Smart and mobile embedded web server," in *Complex, Intelligent and Software Intensive Systems, 2009. CISIS '09. International Conference on*, 2009.

[93] B. Villaverde, D. Pesch, R. De Paz Alberola, S. Fedor, and M. Boubekeur, "Constrained application protocol for low power embedded networks : A survey," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, 2012.

[94] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific American*, vol. 284, no. 5, 2001.

[95] M. Kifer, "Rule interchange format : The framework," in *Web Reasoning and Rule Systems*, Springer Berlin Heidelberg, 2008.

[96] S. Hachem, T. Teixeira, and V. Issarny, "Ontologies for the internet of things," in *Proceedings of the 8th Middleware Doctoral Symposium*, 2011.

[97] A. Bröring, J. Echterhoff, S. Jirka, I. Simonis, T. Everding, C. Stasch, S. Liang, and R. Lemmens, "New generation sensor web enablement," *Sensors*, vol. 11, no. 3, 2011.

[98] M. Compton, P. Barnaghi, L. Bermudez, R. Garcıa-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, V. Huang, K. Janowicz, W. D. Kelsey, D. L. Phuoc, L. Lefort, M. Leggieri, H. Neuhaus, A. Nikolov, K. Page, A. Passant, A. Sheth, and K. Taylor, "The SSN ontology of the W3C semantic sensor network incubator group," *Web Semantics : Science, Services and Agents on the World Wide Web*, 2012.

[99] S. Nath, A. Deshpande, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan, "IrisNet : An architecture for internet-scale sensing services," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, 2003.

[100] J. Campbell, P. B. Gibbons, S. Nath, P. Pillai, S. Seshan, and R. Sukthankar, "IrisNet : An internet-scale architecture for multimedia sensors," in *Proceedings of the 13th Annual ACM International Conference on Multimedia*, MULTIMEDIA '05, 2005.

[101] K.-C. Chen and S.-Y. Lien, "Machine-to-machine communications : Technologies and challenges," *Ad Hoc Networks*, vol. 18, no. 0, 2014.

[102] F. Delicato, P. Pires, and T. Batista, "Smartsensor : An infrastructure for the web of things," in *Middleware Solutions for the Internet of Things*, Springer London, 2013.

[103] S. Distefano, G. Merlino, and A. Puliafito, "Enabling the cloud of things," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, 2012.

[104] R. Petrolo, V. Loscrí, and N. Mitton, "Towards a smart city based on cloud of things," in *Proceedings of the 2014 ACM International Workshop on Wireless and Mobile Technologies for Smart Cities*, WiMobCity '14, 2014.

[105] S. Distefano, G. Merlino, and A. Puliafito, "Towards the cloud of things sensing and actuation as a service, a key enabler for a new cloud paradigm," in *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on*, 2013.

[106] M. Yuriyama and T. Kushida, "Sensor-cloud infrastructure - physical sensor management with virtualized sensors on cloud computing," in *Network-Based Information Systems (NBiS), 2010 13th International Conference on*, 2010.

[107] D. Alessandrelli, M. Petraccay, and P. Pagano, "T-res : Enabling reconfigurable in-network processing in iot-based wsns," in *Proceedings of the 2013 IEEE International Conference on Distributed Computing in Sensor Systems*, 2013.

[108] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin, "A wireless sensor network for structural monitoring," in *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, 2004.

[109] F. Thiesse, C. Floerkemeier, M. Harrison, F. Michahelles, and C. Roduner, "Technology, standards, and real-world deployments of the EPC network," *Internet Computing, IEEE*, vol. 13, no. 2, 2009.

[110] F. Sailhan, J. Bourgeois, and V. Issarny, "A security supervision system for hybrid networks," in *Software Engineering, Artificial Intelligence, Networking and Parallel/-Distributed Computing*, Springer, 2008.

[111] R. K. Rana, C. T. Chou, S. S. Kanhere, N. Bulusu, and W. Hu, "Ear-phone : An end-to-end participatory urban noise mapping system," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2010.

[112] M. Sharifi, S. Kafaie, and O. Kashefi, "A survey and taxonomy of cyber foraging of mobile devices," *Communications Surveys Tutorials, IEEE*, vol. 14, no. 4, 2012.

[113] T. Verbelen, P. Simoens, F. De Turck, and B. Dhoedt, "Cloudlets : Bringing the cloud to the mobile user," in *Proceedings of the Third ACM Workshop on Mobile Cloud Computing and Services*, 2012.

[114] M. Satyanarayanan, "The role of cloudlets in hostile environments," in *Proceeding of the Fourth ACM Workshop on Mobile Cloud Computing and Services*, 2013.

[115] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, 2012.

[116] M. C. Domingo, "An overview of the internet of underwater things," *Journal of Network and Computer Applications*, vol. 35, no. 6, 2012.

[117] I. Akyildiz and J. Jornet, "The internet of nano-things," *Wireless Communications, IEEE*, vol. 17, no. 6, 2010.

[118] A. Arasu, S. Babu, and J. Widom, "CQL : A language for continuous queries over streams and relations," in *Database Programming Languages*, Springer Berlin Heidelberg, 2004.

[119] D. Terry, D. Goldberg, D. Nichols, and B. Oki, "Continuous queries over append-only databases," *SIGMOD Rec.*, vol. 21, no. 2, 1992.

[120] P. Seshadri, M. Livny, and R. Ramakrishnan, "$\mathcal{SEQ}$: A model for sequence databases," in *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, 1995.

[121] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, 1970.

[122] A. J. Symonds and R. A. Lorie, "A schema for describing a relational data base," in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, (New York, NY, USA), ACM, 1970.

[123] L. Golab and M. T. Özsu, "Issues in data stream management," *SIGMOD Rec.*, vol. 32, no. 2, 2003.

[124] Y.-N. Law, H. Wang, and C. Zaniolo, "Query languages and data models for database sequences and data streams," in *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, VLDB Endowment, 2004.

[125] A. V. Aho and J. D. Ullman, "Universality of data retrieval languages," in *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, (New York, NY, USA), ACM, 1979.

[126] H. Wang, C. Zaniolo, and C. R. Luo, "ATLAS : A small but complete SQL extension for data mining and data streams," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, VLDB Endowment, 2003.

[127] M. Sullivan, "Tribeca : A stream database manager for network traffic analysis," in *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, (San Francisco, CA, USA), Morgan Kaufmann Publishers Inc., 1996.

[128] S. Babu and J. Widom, "Continuous queries over data streams," *SIGMOD Rec.*, vol. 30, no. 3, 2001.

[129] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "STREAM : The stanford data stream management system," tech. rep., Stanford InfoLab, 2004.

[130] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul, "SECRET : A model for analysis of the execution semantics of stream processing systems," *Proc. VLDB Endow.*, vol. 3, no. 1-2, 2010.

[131] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik, "Towards a streaming SQL standard," *Proc. VLDB Endow.*, vol. 1, no. 2, 2008.

[132] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora : A new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, 2003.

[133] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik, "High-availability algorithms for distributed stream processing," in *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, (Washington, DC, USA), IEEE Computer Society, 2005.

[134] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the borealis stream processing engine," in *In CIDR*, 2005.

[135] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language : Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, 2006.

[136] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik, "Providing resiliency to load variations in distributed stream processing," in *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, VLDB Endowment, 2006.

[137] C. Lei and E. A. Rundensteiner, "Robust distributed query processing for streaming data," *ACM Trans. Database Syst.*, vol. 39, no. 2, 2014.

[138] Y. Xing, S. Zdonik, and J.-H. Hwang, "Dynamic load distribution in the borealis stream processor," in *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, (Washington, DC, USA), IEEE Computer Society, 2005.

[139] P. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 15, May 2003.

[140] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing : A new architecture for high-performance stream systems," *Proc. VLDB Endow.*, vol. 1, no. 1, 2008.

[141] U. Srivastava and J. Widom, "Flexible time management in data stream systems," in *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '04, 2004.

[142] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck, "A heartbeat mechanism and its application in gigascope," in *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, 2005.

[143] R. Fernández-Moctezuma, K. Tufte, and J. Li, "Inter-operator feedback in data stream management systems via punctuation," in *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research (CIDR)*, CIDR' 09, 2009.

[144] R. V. Nehme, H.-S. Lim, and E. Bertino, "FENCE : Continuous access control enforcement in dynamic data stream environments," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, 2013.

[145] W. Wolf, *Computers As Components : Principles of Embedded Computing System Design.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2nd ed., 2008.

[146] S. Sakr, A. Liu, and A. G. Fayoumi, "The family of Mapreduce and large-scale data processing systems," *ACM Comput. Surv.*, vol. 46, no. 1, 2013.

[147] J. Gregorio and B. de Hora, "RFC 5023 - datagram transport layer security version 1.2." *http://tools.ietf.org/html/rfc5023* (accédé le 03/09/2014), 2012.

[148] D. Guinard, V. Trifa, and E. Wilde, "A resource oriented architecture for the web of things," in *Internet of Things (IOT), 2010*, 2010.

[149] J. Schneider, T. Kamiya, D. Peintner, and R. Kyusakov, "Efficient XML interchange (EXI) format 1.0 (second edition)." *http://www.w3.org/TR/exi* (accédé le 03/09/2014), 2014.

[150] I. Fette and A. Melnikov, "RFC 6455 - the WebSocket protocol." *http://tools.ietf.org/html/rfc6455* (accédé le 03/09/2014), 2011.

[151] V. Pimentel and B. Nickerson, "Communicating and displaying real-time data with websocket," *Internet Computing, IEEE*, vol. 16, no. 4, 2012.

[152] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, "SOAP version 1.2 part 1 : Messaging framework (second edition)." *http://www.w3.org/TR/2007/REC-soap12-part1-20070427/* (accédé le 03/09/2014), 2007.

[153] G. Lam and D. Rossiter, "A web service framework supporting multimedia streaming," *Services Computing, IEEE Transactions on*, vol. 6, no. 3, 2013.

[154] M. Ruth, F. Lin, and S. Tu, "Adapting single-request/multiple-response messaging to web services," in *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, 2005.

[155] K. Hartke, "Observing resources in CoAP." *http://www.ietf.org/id/draft-ietf-core-observe-15.txt* (accédé le 03/09/2014), 2014.

[156] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns : elements of reusable object-oriented software.* Addison-Wesley, 1995.

[157] C. Bormann and Z. Shelby, "Blockwise transfers in CoAP." *http://www.ietf.org/id/draft-ietf-core-block-16.txt* (accédé le 03/09/2014), 2014.

[158] "Le système SI d'unités de mesure." *http://www.entreprises.gouv.fr/metrologie/systeme-si-dunites-mesure* (accédé le 03/09/2014), 2012.

[159] N. Freed and N. Borenstein, "RFC 2046 - multipurpose internet mail extensions (MIME) part two : Media types." *http://tools.ietf.org/html/rfc2046* (accédé le 03/09/2014), 1996.

[160] N. Ahmed, M. Linderman, and J. Bryant, "Towards mobile data streaming in service oriented architecture," in *Reliable Distributed Systems, 2010 29th IEEE Symposium on*, 2010.

[161] M. Valipour, B. Amirzafari, K. Maleki, and N. Daneshpour, "A brief survey of software architecture concepts and service oriented architecture," in *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, 2009.

[162] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Commun. ACM*, vol. 3, no. 4, 1960.

[163] M. Sperber, R. K. Dybvig, M. Flatt, A. Van Straaten, R. Findler, and J. Matthews, "Revised report on the algorithmic language scheme," *Journal of Functional Programming*, vol. 19, 2009.

[164] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance : Building a better Bloom filter," *Random Struct. Algorithms*, vol. 33, no. 2, 2008.

[165] O. Maye and M. Maaser, "Comparing java virtual machines for sensor nodes," in *Grid and Pervasive Computing*, Springer, 2013.

[166] L. Masinter, "RFC 2388 : Returning values from forms : multipart/form-data." *http://tools.ietf.org/html/rfc2388* (accédé le 03/09/2014), 1998.

[167] D. Crocker and P. Overell, "RFC 5234 - augmented BNF for syntax specifications : ABNF." *http://tools.ietf.org/html/rfc5234* (accédé le 12/01/2014), 2008.

[168] N. Anciaux, M. Benzine, L. Bouganim, P. Pucheral, and D. Shasha, "Revelation on demand," *Distributed and Parallel Database (Weston, Conn.) Journal*, vol. 25, no. 1-2, 2009.

[169] N. Li, N. Zhang, S. K. Das, and B. Thuraisingham, "Privacy preservation in wireless sensor networks : A state-of-the-art survey," *Ad Hoc Networks*, vol. 7, no. 8, 2009.

[170] A. Wilschut and P. M. G. Apers, "Pipelining in query execution," in *Proc. of the International Conference on Databases, Parallel Architectures and Their Applications*, PARBASE '90, 1990.

[171] Y. Yi, W. Han, X. Zhao, A. Erdogan, and T. Arslan, "An ILP formulation for task mapping and scheduling on multi-core architectures," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, 2009.

[172] J. Teich, "Hardware/software codesign : The past, the present, and predicting the future," *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, 2012.

[173] X. Grehant, I. Demeure, and S. Jarp, "A survey of task mapping on production grids," *ACM Computing Surveys*, vol. 45, no. 3, 2013.

[174] A. Pathak and V. Prasanna, "Energy-efficient task mapping for data-driven sensor network macroprogramming," *IEEE Transactions on Computers*, vol. 59, no. 7, 2010.

[175] S. Bokhari, "On the mapping problem," *IEEE Transactions on Computers*, vol. 30, no. 3, 1981.

[176] P. K. Sahu and S. Chattopadhyay, "A survey on application mapping strategies for network-on-chip design," *Journal of Systems Architecture*, vol. 59, no. 1, 2013.

[177] J. W. Chinneck, "Practical optimization : a gentle introduction." *http://www.sce.carleton.ca/faculty/chinneck/po.html* (accédé le 03/09/2014), 2012.

[178] K. Sörensen, "Metaheuristics – the metaphor exposed," *International Transactions in Operational Research*, 2013.

[179] N. Edalat, C.-K. Tham, and W. Xiao, "An auction-based strategy for distributed task allocation in wireless sensor networks," *Computer Communications*, vol. 35, no. 8, 2012.

[180] Z. Lu, Y. Wen, R. Fan, S.-L. Tan, and J. Biswas, "Toward efficient distributed algorithms for in-network binary operator tree placement in wireless sensor networks," *IEEE Selected Areas in Communications*, vol. 31, no. 4, 2013.

[181] F. H. Bijarbooneh, P. Flener, E. Ngai, and J. Pearson, "Energy-efficient task-mapping for data-driven sensor network macroprogramming using constraint programming," in *Proc. of the 9th International Workshop on Constraint Modelling and Reformulation*, 2010.

[182] D. S. Johnson, "The NP-completeness column," *ACM Trans. Algorithms*, vol. 1, no. 1, 2005.

[183] T. C. Koopmans and M. Beckmann, "Assignment problems and the location of economic activities," *Econometrica : journal of the Econometric Society*, 1957.

[184] G. J. Woeginger, "Exact algorithms for NP-Hard problems : A survey," in *Combinatorial Optimization — Eureka, You Shrink !*, Springer Berlin Heidelberg, 2003.

[185] J. Hart and A. W. Shogan, "Semi-greedy heuristics : An empirical study," *Operations Research Letters*, vol. 6, no. 3, 1987.

[186] K. A. Dowsland, "Genetic algorithms," in *Search Methodologies*, Springer US, 2005.

[187] R. Bosch and M. Trick, "Genetic algorithms," in *Search Methodologies*, Springer US, 2005.

[188] M. Gendreau and J.-Y. Potvin, "Tabu search," in *Search Methodologies*, Springer US, 2005.

[189] O. Buffet, L. Cucu, L. Idoumghar, and R. Schott, "Tabu search type algorithms for the multiprocessor scheduling problem," in *10th IASTED International Conference on Artificial Intelligence and Applications - AIA 2010*, 2010.

[190] P. Yan and C. Zheng, "Evolutionary tabu search in task allocation of unmanned aerial vehicles," in *Computational Engineering in Systems Applications, IMACS Multiconference on*, 2006.

[191] K. Sastry, D. E. Goldberg, and G. Kendall, "Genetic algorithms," in *Search Methodologies*, Springer US, 2005.

[192] A. Tengg, A. Klausner, and B. Rinner, "Task allocation in distributed embedded systems by genetic programming," in *Parallel and Distributed Computing, Applications and Technologies, 2007. PDCAT '07. Eighth International Conference on*, 2007.

[193] F. A. Omara and M. M. Arafa, "Genetic algorithms for task scheduling problem," *Journal of Parallel and Distributed Computing*, vol. 70, no. 1, 2010.

[194] S. Pandey, L. Wu, S. Guru, and R. Buyya, "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments," in *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, 2010.

[195] F. Ferrandi, P. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo, "Ant colony optimization for mapping, scheduling and placing in reconfigurable systems," in *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on*, 2013.

[196] W. Heinzelman, A. Sinha, A. Wang, and A. Chandrakasan, "Energy-scalable algorithms and protocols for wireless microsensor networks," in *Proc. of the 25th IEEE International Conference on Speech, and Signal Processing*, 2000.

[197] D. Schmidt, M. Krämer, T. Kuhn, and N. Wehn, "Energy modelling in sensor networks," *Advances in Radio Science*, vol. 5, no. 12, 2007.

[198] S. Malek, N. Medvidovic, and M. Mikic-Rakic, "An extensible framework for improving a distributed software system's deployment architecture," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, 2012.

[199] M. Bhatti, C. Belleudy, and M. Auguin, "An inter-task real time DVFS scheme for multiprocessor embedded systems," in *Proc. of 4th the Conference on Design and Architectures for Signal and Image Processing*, 2010.

[200] W. Li, F. C. Delicato, and A. Y. Zomaya, "Adaptive energy-efficient scheduling for hierarchical wireless sensor networks," *ACM Trans. Sen. Netw.*, vol. 9, no. 3, 2013.

[201] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proc. of the Workshop on Experimental Computer Science*, 2007.

[202] M. Junger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleybank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, *50 Years of Integer Programming 1958-2008*. Springer, 2010.

[203] M. Demirbas, "Wireless Sensor Networks for Monitoring of Large Public Buildings," tech. rep., Univ. at Buffalo, 2005.

[204] H. Qin, Y. Wang, and W. Zhang, "ZigBee-assisted WiFi transmission for multi-interface mobile devices," in *Mobile and Ubiquitous Systems : Computing, Networking, and Services*, Springer Berlin Heidelberg, 2012.

[205] Y. Wu, Q. Z. Sheng, D. Ranasinghe, and L. Yao, "PeerTrack : A platform for tracking and tracing objects in large-scale traceability networks," in *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, 2012.

[206] "IEEE healthcare IT standards." *http://standards.ieee.org/findstds/standard/healthcare_it.html* (accédé le 03/09/2014).

[207] J. Paek, J. Kim, and R. Govindan, "Energy-efficient rate-adaptive GPS-based positioning for smartphones," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, 2010.

[208] A. Khan, Y. K. Lee, S. Lee, and T. S. Kim, "Human activity recognition via an accelerometer-enabled-smartphone using kernel discriminant analysis," in *Proceedings of the 5th International Conference on Future Information Technology*, 2010.

[209] S. Hemminki, P. Nurmi, and S. Tarkoma, "Accelerometer-based transportation mode detection on smartphones," in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, 2013.

[210] M. Papandrea, M. Zignani, S. Gaito, S. Giordano, and G. Rossi, "How many places do you visit a day ?," in *Proceedings of the 11th IEEE International Conference on Pervasive Computing and Communications Workshops*, 2013.

[211] T. Yoon and J.-H. Lee, "Goal and path prediction based on user's moving path data," in *Proceedings of the 2nd International Conference on Ubiquitous Information Management and Communication*, 2008.

[212] X. Lu, E. Wetter, N. Bharti, A. J. Tatem, and L. Bengtsson, "Approaching the limit of predictability in human mobility," *Scientific Reports*, vol. 3, no. 2923, 2013.

[213] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma, "Mining interesting locations and travel sequences from GPS trajectories," in *Proceedings of the 18th International Conference on World Wide Web*, 2009.

Annexes

1 Implémentation d'un filtre de Bloom en *DiSPL*

```
; définition du contrat
(contract (name "bloom-filter")
    (input-ports
        ; un port pour ajouter des éléments dans le filtre
        (port (name "add-port") (min 1) (max 1))
        ; un port pour recevoir les éléments dont la présence dans le filtre doit être testée
        (port (name "test-port") (min 1) (max 1))
        ; schéma identique sur les deux ports
        (accept-rules
            (input-rule (type "same-schema")
                (ports ("add-port" "test-port"))
            )
        )
    )
    (output-ports
        ; un port de sortie pour relayer les éléments s'ils sont présents dans le filtre
        (port (name "tested-item")
            (standard-output 1)
            (output-rule (type "identity")
                (port "test-port")
            )
        )
        ; un port de sortie pour la probabilité de faux négatifs
        (port (name "false-negative-probability")
            (output-rule (type "schema")
                (schema
                    (attribute (name "probability")
                        (concrete-type "primitive/real")
                    )
                )
            )
        )
    )
)

; section d'initialisation
init:
    (let bitsetSize 32)
    (let bitset (bitword bitsetSize false))
    (let nbBuckets 8)
    (let nbItems 0)

; section de travail
work:
    ; tout d'abord, le filtre de Bloom est mis à jour avec le prochain élément
    (let nextItem (get-next-item "add-port"))
    (cond ((notnull nextItem)
        (let hash (murmur3 128 2 nextItem))
        (for 0 to nbBuckets
            ((let index (% (abs (+ (get! hash 0) (* i (get! hash 1)))) bitsetSize))
            (set! bitset index true))
        )
        (increment nbItems 1)

        ; écrit la probabilité de faux négatif dans la sortie correspondante
        (let p (pow (- 1 (exp (* (- nbBuckets) (/ nbItems bitsetSize)))) nbBuckets))
        (write "false-negative-probability"
            (item (timestamp (now)) ("probability" p))
        )
```

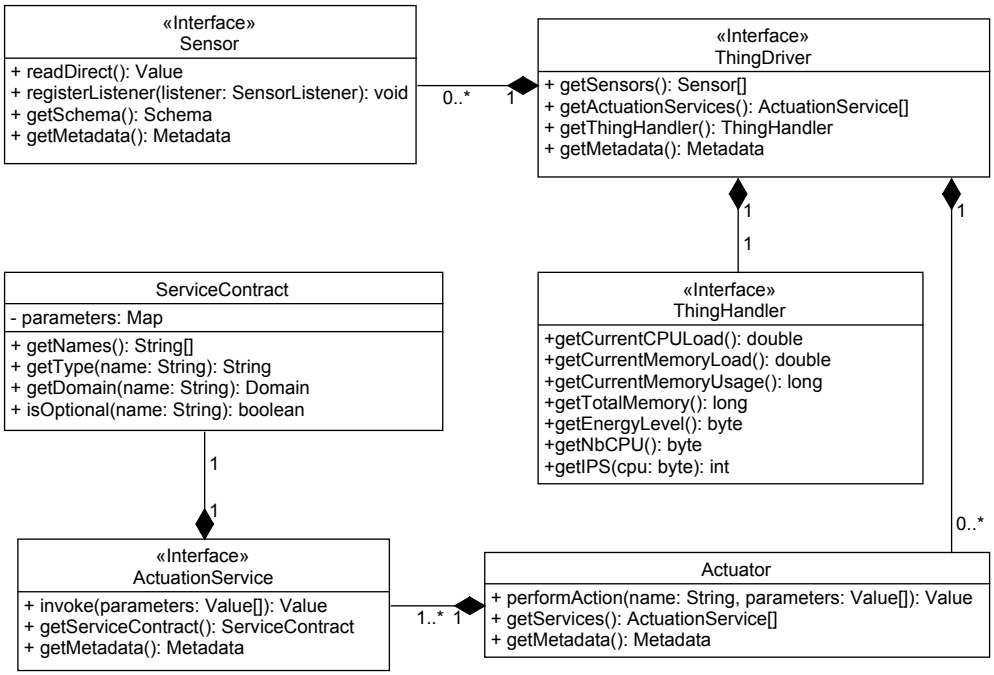
```
60      ))
61
62      ; puis, effectue le test d'appartenance si nécessaire
63      (let request (get-next-item "test-port"))
64      (cond ((notnull request)
65          (let hash (murmur3 128 2 request))
66          (let present true)
67          (for 0 to nbBuckets
68              ((define index (% (abs (+ (get! hash 0) (* i (get! hash 1)))) bitsetSize))
69              (cond ((not (get! bitset index))
70                  (set! present 0)
71                  (break))
72              ))
73          )
74          (if present
75              (write "tested-item" request)
76          )
77      ))
```

Cet exemple de transformateur implémente un filtre de Bloom [1] au moyen du langage *DiSPL*. Le programme admet deux flux en entrée, un flux d'éléments à ajouter au filtre (port « add-port ») et un flux d'éléments dont l'appartenance à l'ensemble doit être testée (port « test-port »). Un élément reçu sur « test-port » est alors écrit sur le port de sortie standard « tested-item » s'il passe le test d'appartenance, ou est rejeté. Lorsque les éléments reçus sur « add-port » sont ajoutés au filtre, la probabilité de faux positifs est alors recalculée puis écrite en tant que nouvel élément sur le port de sortie « false-negative-probability ».

1. Un filtre de Bloom est une structure de donnée légère permettant de vérifier avec certitude la non-appartenance d'un élément à un ensemble. À l'inverse, l'opération d'appartenance est sujette aux faux positifs, avec une probabilité qui croît avec le nombre d'éléments uniques.

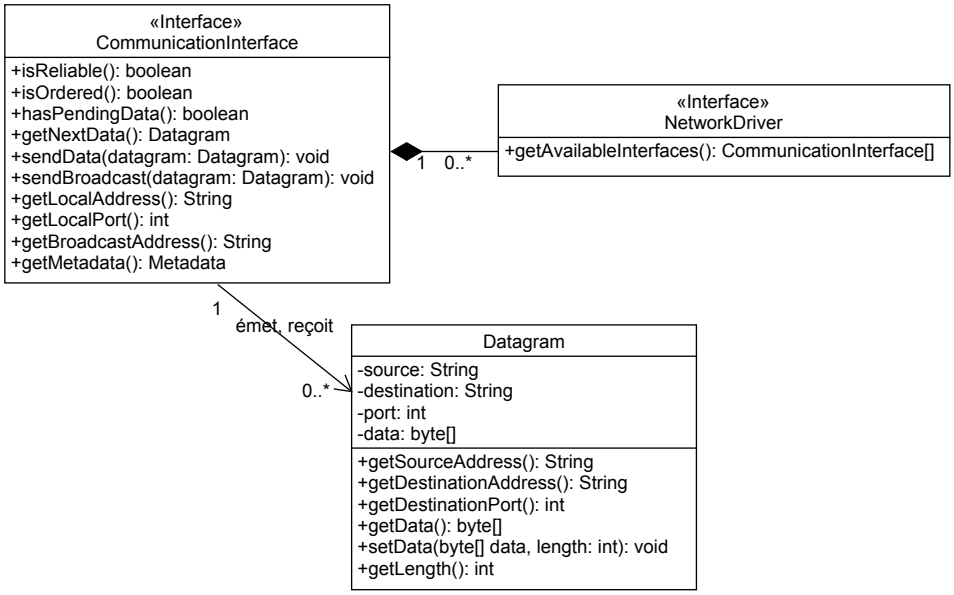
2 Interfaces des pilotes de *Dioptase*

Pilote d'objet



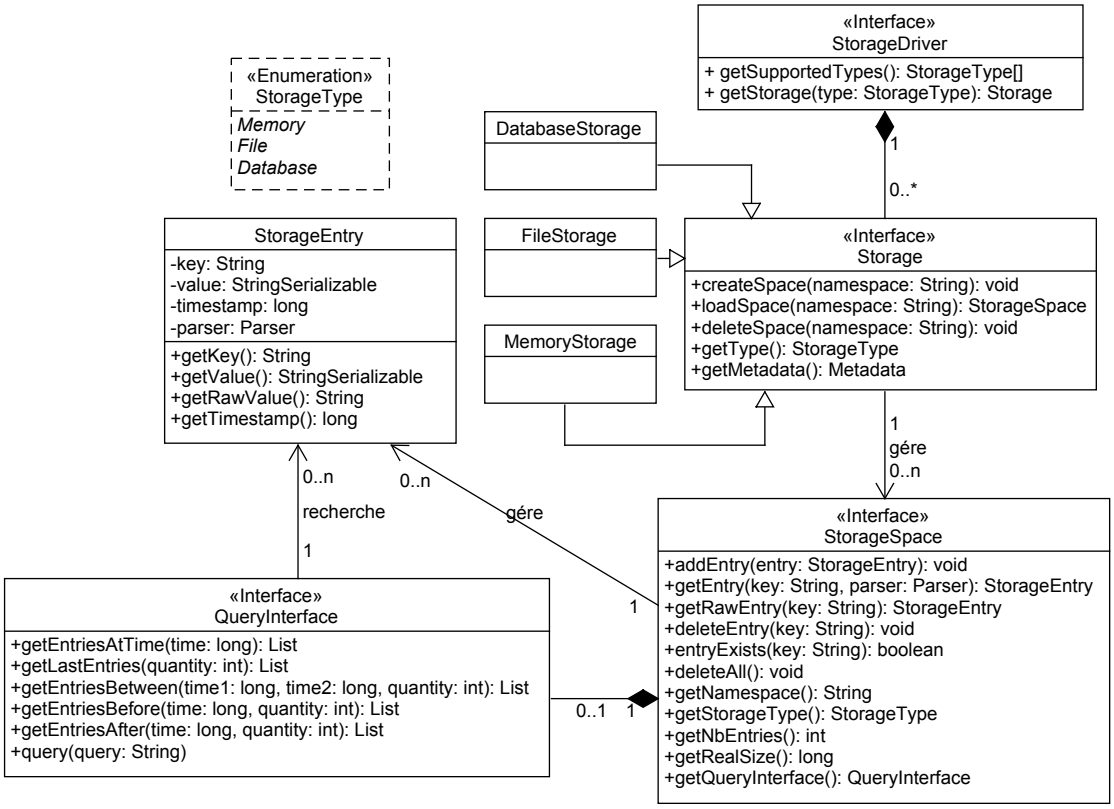
Le pilote d'objet permet d'encapsuler la logique de lecture des capteurs, sous la forme de méthodes d'accès direct (lecture de l'état du capteur) ou d'accès aux derniers évènements générés. Le pilote d'objet permet, en outre, de représenter chaque actionneur sous la forme d'un ou de plusieurs services d'action, chacun d'eux étant décrit par un nom et un ensemble de paramètres spécifiés par un contrat d'actionneur. Pour finir, c'est au travers de ce pilote que *Dioptase* accède aux métadonnées propres à l'objet, sous la forme de paires clés-valeurs, le développeur étant libre de fournir ces informations manuellement ou d'écrire la logique permettant de les collecter automatiquement.

Pilote réseau



Le pilote réseau permet d'échanger des informations avec d'autres appareils et se compose d'une ou de plusieurs interfaces de communication permettant de lire et d'écrire des données de manière asynchrone. Concrètement, la notion de connexion n'existe pas au niveau de *Dioptase* et le pilote doit prendre en charge ces éventuels aspects, selon le protocole de transport considéré dans les différentes interfaces de communication. Comme le pilote est conçu pour représenter des communications au-dessus de la couche transport, chaque bloc de donnée lu ou écrit est associé à une adresse source, une adresse de destination et un numéro de port.

Pilote de stockage



Le pilote de stockage permet d'encapsuler la logique d'accès aux différents espaces de stockage disponibles sur l'objet. Chaque espace de stockage est représenté de manière générique sous la forme de couples clés-valeurs organisés au moyen d'espaces de noms, chaque couple étant associé à un timestamp. Le pilote liste les types de stockage disponible et permet à *Dioptase* et aux applications de les instancier à la volée puis d'y lire ou d'y écrire de nouveaux couples. Notre implémentation fournit trois types communs de stockage : stockage en mémoire (volatile), stockage persistant sous forme de fichiers et stockage persistant sous forme de base de données embarquée.

3 Protocoles de services Web implémentés par *Dioptase*

	HTTP	**CoAP**	**Websocket**
Chiffrement	*plugin* (*TLS*)	*plugin* (*DTLS*)	*plugin* (*TLS*)
Compression	*plugin* (*gzip*, *deflate*)	non prévu	*plugin* (*deflate*)
Respect de l'ordre	implicite (*TCP*)	*plugin*	implicite (*TCP*)
Fiabilité	implicite (*TCP*)	*plugin*	implicite (*TCP*)
Négociation de contenu	non implémenté	non implémenté	non implémenté
Mise en cache	*plugin*	*plugin*	*plugin*
Streaming synchrone	pris en charge (*HTTP streaming*)	non prévu	pris en charge
Streaming asynchrone	pris en charge (*Web hooks*)	pris en charge (*Observable*)	non prévu
Encodages acceptés	*Unicode*	*Unicode*	*Unicode*
Langages acceptés	*en-US*	*en-US*	*en-US*
Formats acceptés	*JSON*, binaire	binaire	*JSON*, binaire
Contrôle d'accès	non implémenté	non implémenté	non implémenté
Cookie	non implémenté	non prévu	non prévu
Connexions réutilisable	oui	non applicable	non
Contrôle d'origine	inactif	non prévu	inactif

pris en charge : fourni dans le cœur de fonctionnalités.
plugin : fourni en tant que *plugin*.
non prévu : ne fait pas partie du protocole.
non implémenté : fait partie du protocole, mais n'est pas fourni par *Dioptase*.

4 Services de gestion implémentés par *Dioptase*

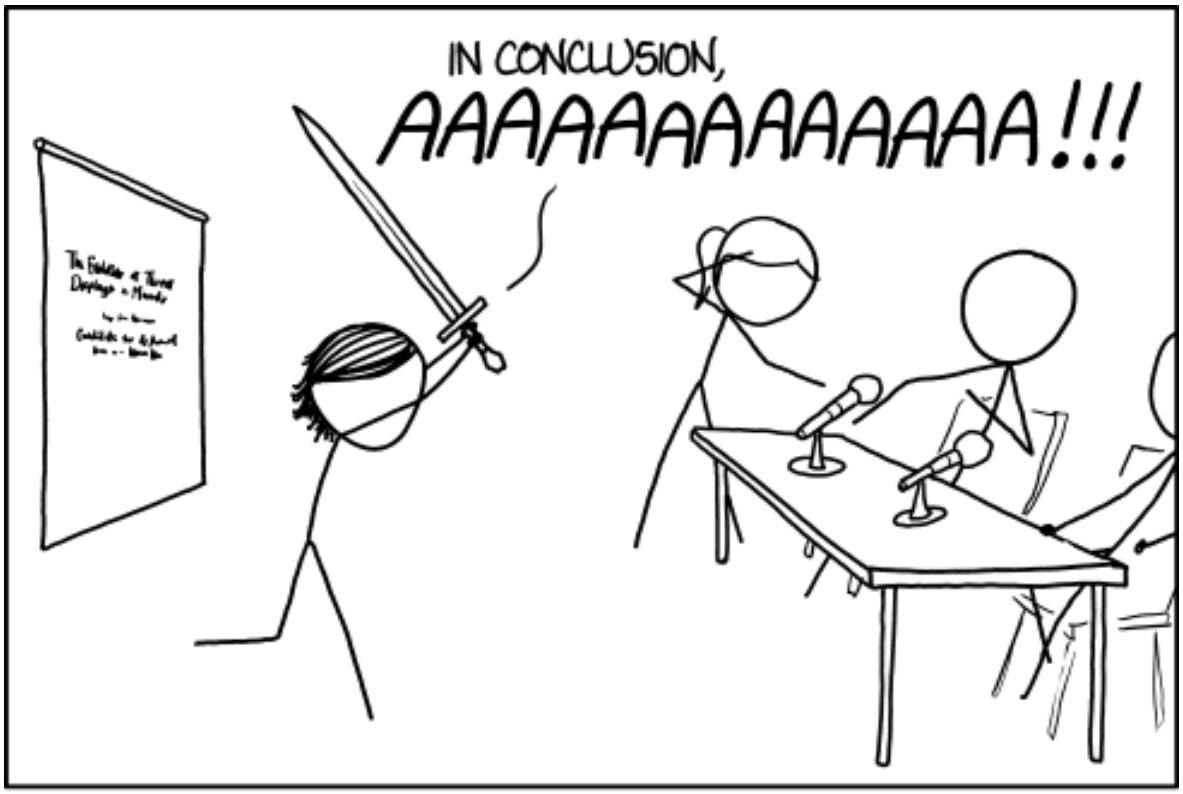
Service	Méthode	Description et paramètres
/thing/properties	GET	Retourne les propriétés de l'objet : capteurs, actionneurs, types de stockage et métadonnées. **sensors=true|false, *actuators=true|false, *storages=true|false, *metadata=true|false*
/thing/sensors	GET	Retourne les noms, les types concrets et sémantiques, les unités de mesure et les métadonnées d'un ou de tous les capteurs disponibles. **id=<nom du capteur>*
/thing/actuators	GET	Retourne les noms et les paramètres d'un ou de tous les actionneurs disponibles. **id=<nom de l'actionneur>*
/thing/sensors/data	GET	Prend une mesure sur un capteur virtuel. Sans effet sur un capteur produisant des évènements. *id=<nom du capteur>*
/thing/actuate	GET	Invoque un service fourni par un actionneur. *id=<nom de l'actionneur>, service=<nom du service>, paramètres spécifiques à l'actionneur (paires clé-valeur)*
/thing/operators	GET	Retourne les noms et les contrats d'un ou de tous les traitements compilés disponibles. **id=<nom du traitement compilé>*
/thing/storages/data	GET	Récupère une donnée identifiée par sa clé depuis un stockage virtuel, ou récupère un sous-ensemble de donnée qui satisfait des contraintes positionnelles ou temporelles. *namespace=<espace de nom>, *key=<clé>, *t-start=<timestamp de départ>, *t-end=<timestamp de fin>|now, *nb=<nombre d'éléments à récupérer>, paramètres spécifiques à l'implémentation du stockage (paires clé-valeur)*
/services	GET	Retourne le nom, le contrat et les *URI* des flux d'entrée et de sortie d'un ou de tous les services continus. **id=<nom du service>*
/services	POST	Démarre un nouveau service continu. *id=<nom du service>*, paramètres spécifiques au service
/services	DELETE	Arrête et détruit un service continu. *id=<nom du service>*
/processors	GET	Retourne le nom et le schéma des flux d'entrée et de sortie d'un ou de tous les transformateurs. Peut aussi retourner des informations étendues comme l'opération exécutée (nom du transformateur ou code *DiSPL*) et le contenu de l'état interne. *id*=<nom du transformateur>, *extended=true|false*
/processors	POST	Déploie un nouveau transformateur. Par défaut, celui-ci n'est pas démarré automatiquement (sauf si *autostart=true*). *id=<nom du transformateur>, code=<code DiSPL> ou operator=<nom de traitement compilé>, inputs=<URI des flux d'entrée>, *autostart=true|false, *state=<état de démarrage>, *history=true|false, *h-size=<taille maximale de l'historique>, paramètres spécifiques à l'opération (paires clé-valeur)*
/processors/start	GET	Démarre un transformateur précédemment déployé. *id=<nom du transformateur>*, *priority=<priorité d'exécution>*
/processors/state	GET	Retourne le contenu de l'état interne d'un transformateur. *id=<nom du transformateur>*

/processors/history	GET	Retourne l'historique du flux de sortie d'un transformateur. *id=<nom du transformateur>, *t-start=<timestamp de départ>, *t-end=<timestamp de fin>|now, *nb=<nombre maximum d'éléments>*
/processors/migrate	GET	Migre un transformateur sur un autre appareil, avec ou sans son état interne. *id=<nom du transformateur>, to=<URI de l'appareil destinataire>, *forget-state=true|false*
/producers	GET	Retourne le nom, la source de données et le schémas du flux de sortie d'un ou de tous les producteurs. *id*=<nom du producteur>*
/producers	POST	Déploie un nouveau producteur. *id=<nom du producteur>, source=<nom de la source>, paramètres spécifiques au type de producteur (paires clé-valeur)*
/producers/history	GET	Retourne l'historique du flux de sortie d'un producteur. *id=<nom du producteur>, *t-start=<timestamp de départ>, *t-end=<timestamp de fin>|now, *nb=<nombre maximum d'éléments>*
/storages	GET	Retourne le nom, le type et le schéma des flux d'entrée et de sortie d'un ou de tous les stockages. *id*=<nom du stockage>*
/storages	POST	Déploie un nouveau stockage. *id=<nom du stockage>, type=<type de stockage>, inputs=<URI des flux d'entrée>, *size=<taille du stockage en octets>, paramètres spécifiques au type de stockage (paires clé-valeur)*
/consumers	GET	Retourne le nom et la description des consommateurs déployés. *id*=<nom d'un consommateur>*
/consumers	POST	Déploie un nouveau consommateur. *id=<nom du consommateur>, actuation=<nom du service d'action>, inputs=<URI des flux d'entrée>, paramètres spécifiques à l'implémentation du consommateur (paires clé-valeur)*

abcdef : paramètre optionnel.

<abcdef> : admet une valeur quelconque.

x|*y*|*z* : admet la valeur *x*, *y* ou *z*.



THE BEST THESIS DEFENSE IS A GOOD THESIS OFFENSE.

Ceci est la fin de ce manuscrit.

Tout texte qui suivrait cette mention signifierait (i) que le présent document a été altéré par des individus ou organisations aux intentions inconnues, (ii) qu'une erreur est survenue lors de l'agrafage ou (iii) qu'il est probablement temps de mettre de l'ordre sur votre bureau.